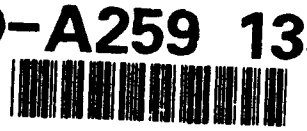


AFIT/GCS/ENG/92D-07

VERSION AND TRANSACTION MANAGEMENT IN OSMAGIC: AN OBJECT-ORIENTED DATABASE IMPLEMENTATION OF THE MAGIC VLSI LAYOUT DESIGN TOOL

THESIS

Gary Michael Lightner
Captain, USAF

AFIT/GCS/ENG/92D-07

AFIT/GCS/ENG/92D-07

VERSION AND TRANSACTION MANAGEMENT IN OSMAGIC: AN OBJECT-ORIENTED DATABASE IMPLEMENTATION OF THE MAGIC VLSI LAYOUT DESIGN TOOL

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

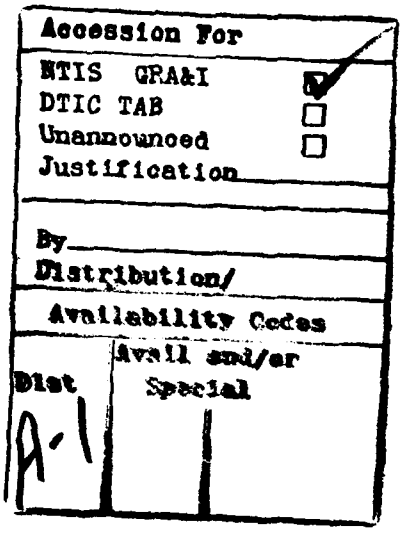
Requirements for the Degree of

Master of Science

Gary Michael Lightner, B.S.

Captain, USAF

December, 1992



Acknowledgements

No person can do work of this magnitude without the help and support of others. This thesis is no exception. I owe a great deal of thanks and gratitude to my faculty advisor, Maj Mark A. Roth for his advise and patience in helping me through the fog which was very thick at times. I am also deeply indebted to Mr. Ken Rugg of the Object Design customer support section for his never ending willingness to answer my questions concerning the ObjectStore DBMS and more. I must also thank my fellow students who were always there when I needed to just talk my way through problems and concerns, especially Don, Dave, and Jeff. Most of all, I must thank my wife Rosa and my children José and Rosa Marie for their undying patience and understanding for all the days and nights when the books and research came before them. I could not have completed this work without their love, prayers, and support.

Gary Michael Lightner

Table of Contents

List of Figures

List of Tables

Abstract

The goal of this thesis was to study the feasibility of using an object-oriented database management system to provide the functionality and performance needed to support a complex computer-aided design tool. We do this by modifying OSMagic, a prototype system of the Magic very large scale integrated (VLSI) circuit design tool implemented on the ObjectStore object-oriented database management system. OSMagic was changed to support three different interfaces to ObjectStore and to work in a networking environment. We then designed and examined the use of version and transaction management models as a means of addressing the weaknesses of the prototype as well as a means of introducing new multi-user capabilities. Throughout the effort errors discovered in data definition and allocation were corrected and lessons which pertain to all such conversions were noted. While the current OSMagic is not yet a fully functional version of Magic, the foundation is now in place for achieving this by implementing our more complex transaction model. OSMagic performs as well or better than Magic in about half of the areas tested. Those areas that showed performance degradation were examined with respect to the tradeoffs between added multi-user capabilities and database support versus the severity of the degradation, user perception of the degradation, and the frequency of use for each area. When these tradeoffs were considered along with the expected increased performance of a fully implemented and optimized OSMagic, we conclude that it is highly probable that object-oriented database management systems can provide the functionality and performance needed to support typical computer-aided design applications.

VERSION AND TRANSACTION MANAGEMENT IN OSMAGIC: AN OBJECT-ORIENTED DATABASE IMPLEMENTATION OF THE MAGIC VLSI LAYOUT DESIGN TOOL

I. Introduction

1.1 Overview

Engineering design organizations typically support some sort of cooperative design environment in which multiple designers work separately or in teams to produce design objects which are unique and complex in nature. Designers in these organizations use engineering applications like computer-aided design (CAD) and computer-aided software engineering (CASE) tools to create and manipulate complex and data intensive designs consisting of thousands of individual data components, with each component containing many complicated relationships between various data items. These components, often referred to as *design* data, are dynamic in nature and can have multiple representations and multiple levels of abstraction. Design environments usually maintain a repository of previously designed components which can be used to build other more complex design objects. Because engineering design objects are so complex, these repositories are usually very large, holding components representing every level of detail found in design objects created by the organization. It is not unusual for several repositories to be maintained, each with design objects in various stages of completion and stability, and each controlled by different people. Such repositories are usually implemented as directories of files, with each file containing data and instructions describing the design component they represent. For example, there may be a directory of files representing approved design components which are considered stable and usable in the creation of other design objects as well as directories of files representing design objects which are in various stages of development and thus considered unstable. Design environments also support separate work areas for designers and teams of designers in which design objects are developed and tested before being placed

in the appropriate repository. These workspaces are also implemented as directories with privileges set which control access to the directories and the files contained in them.

Each design tool used in an environment usually has its own file system which must be manually managed. Structures are established for the data files used by the tool and if the structure is modified in any way the programs which access those files must be modified as well. Because the last version of a data file saved overwrites any previous version, concurrency control and design recovery actions must be manually ensured by the maintainer of the data repositories through the enforcement of mutually agreed upon procedures. Procedures and policies are needed for the sharing of data files which usually involves managing multiple copies of the same data file. A system of access privileges is required to ensure proper control over the changing of data files in the various repositories. If such procedures are not followed faithfully, errors due to deletion or modification of the wrong versions of design components could occur. Because most design tools do not automatically handle the affects of concurrent access of the same data files, they are considered *single-user* tools where data consistency is only guaranteed for the version of a data file accessed by any particular invocation of the tool.

While it seems obvious that a database management system (DBMS) would be beneficial in design environments, very few design tools have been implemented with database support. Database management systems provide a very effective means for managing the large amounts of data common in design environments. They allow multiple users and applications access to the same single repository of data while maintaining data consistency through automatic concurrency control and recovery mechanisms. Because of this, tools implemented with a DBMS are considered *multi-user* tools. Database systems also allow for efficient relationship modeling and for the incorporation of additional applications without knowledge of the physical layout of the data. Management and control features eliminate the need for multiple copies of the same data object and enforce effective data security. The amount of maintenance is greatly reduced through the elimination of the need for multiple directories of data files and the different file structures for each design tool.

Even with all their apparent advantages, database management systems are not commonly incorporated in design applications. There are several reasons why design applications, like computer-aided design tools, have spurned DBMS. First traditional database management systems, based on the relational and network paradigms, typically perform too slow for the complex and data intensive algorithms of typical design applications. Secondly, traditional DBMS are oriented towards business applications where the data is simpler and better behaved. Such systems use records and tables to model this *business* data. Because of this record-oriented approach, traditional database systems cannot effectively model the object-oriented *design* data found in engineering design environments. Also, traditional DBMS are oriented towards the shorter transactions found in business applications and cannot adequately support the need for long duration transactions typical in design applications.

The increased demand for computer-aided design tools has resulted in an increased push for the development of data models which better support the design data these tools manipulate. With the introduction of the object-oriented programming paradigm and the advent of object-oriented database management systems (OODBMS), came the hope that complex design applications would finally be able to enjoy the benefits of database management support. These OODBMS promise support for the complex data types common in design applications through the use of object data structures to model the data instead of the records and tables of traditional DBMS. Also, because the data models used are based on the object-oriented paradigm, these new database systems now support the structural navigation common in the algorithms of design applications as well as the need for long transactions. Most OODBMS provide versioning facilities for maintaining the multiple versions of design objects which are so prevalent in design environments. In addition, the few object-oriented database systems now available commercially promise this support with little or no loss of performance. The advent of these object-oriented database management systems creates the need for research which will determine whether or not such systems can truly satisfy the long awaited need for database support in computer-aided design tools.

This thesis continues the effort started by Jacobs (8) to examine the potential for an object-oriented database management system to support complex computer-aided design tools. The very large scale integrated (VLSI) circuit design tool created in that effort, OSMagic, is improved through the resolution of problems encountered in the allocation of persistent and transient data objects and in the saving of persistent data objects. New versioning and transaction models are implemented in an effort to overcome the tool's functionality and size limitations and to provi'e the foundation for expected multi-user capabilities. The performance of the tool with these changes implemented is compared to the performance of the original tool, Magic, which was implemented with a flat file system. An analysis of these performance results and the problem areas encountered in this effort is presented and conclusions drawn about the lessons learned.

1.2 Background

Magic is a very large scale integrated (VLSI) circuit layout design tool which is used at the Air Force Institute of Technology (AFIT) and other United States installations. OS-Magic is an object-oriented implementation of Magic on ObjectStore, an object-oriented database management system (OODBMS) developed by Object Design, Incorporated. OS-Magic was developed by Capt Tim Jacobs during his research at the Air Force Institute of Technology in 1991 (8) to demonstrate that a complex engineering design system could be implemented on a commercially available OODBMS without loss of performance or functionality. Capt Jacobs was unable to provide conclusive results about such a conversion effort since the initial version of OSMagic was not able to provide the full functionality of Magic nor did it address the expected added capabilities a database should bring to the tool.

Magic is a very complicated application written in C with heavy dependence on macros and bit-mask manipulations. It was written as a flat file system specifically designed for Magic and thus many of its functions are tailored to be most efficient for such a system, even at the expense of structured programming. The main unit of work in Magic is the *cell definition.* Cell definitions are created in accordance with the instructions in the flat file representing a cell or as internal work cells which manipulate copies of the definitions kept

on disk. Flat files are created for each cell which is to be saved and each collection of cells representing VLSI designs. The internal cell definitions last for the life of the process and are recreated each time the tool is used. Management and control of the designs created by the tool is up to the user community. It is their responsibility to insure consistency of designs and to control the effects of multiple users working on the same design or parts of the same design.

The initial OSMagic connected Magic with the ObjectStore OODBMS by persistently allocating the cell definitions created by Magic and the hash tables which track those definitions. It changed the read and write operations so that reads checked the database first before looking on disk and writes were essentially not required. The ObjectStore interface chosen for the effort was ObjectStore's Data Manipulation Language (DML). This required changes to all files which used ObjectStore commands to make those files compatible with C++ since the ObjectStore DML uses a C++ compiler. This first effort was implemented using two database transactions. A small initialization transaction and then a large main transaction which lasted for the rest of the process. As with the original Magic, management and control issues are the responsibility of the user community.

1.3 Problem Statement

OSMagic has severe memory limitations, is not a complete conversion of Magic, and does not take advantage of the added capabilities introduced by the database. OSMagic uses only one big transaction for accessing the database. This limits the size of designs the tool can create or manipulate to the size of main memory since any changes must occur within a transaction. All of Magic's commands do not work in OSMagic. With Magic, a designer can go back to a previous version of a design. This cannot be done with OSMagic. OSMagic does not properly define all data that should be persistent and in some cases defines data structures persistently which should remain transiently defined. For these reasons, OSMagic is not a complete conversion of Magic. If a more complete comparison of the two implementations is to be made, OSMagic must have all the capabilities of Magic. Furthermore OSMagic does not yet address the added capabilities like cooperative design

and other multi-user issues that a complete and proper implementation with a database introduces.

1.4 Research Objectives

The primary objective of this thesis is to determine whether or not an object-oriented database management system can provide the performance and functionality required to support a typical computer-aided design tool. To achieve this goal, changes and additions must be made to OSMagic to make it a fully functional implementation of Magic whose performance remains acceptable to the tool's users.

A secondary objective is to show that the time and effort spent in such a conversion is justified by the increased utility of the tool and the reduction in the tool's future maintenance costs. To achieve this goal a workspace model is developed that implements the cooperative design philosophy and methodology of the local Magic user community.

1.5 Methodology

This research is conducted in five stages. The first stage sets up a new working environment and adjusts the current implementation of OSMagic to that environment. The next three stages each address one of OSMagic's three major shortcomings. The fifth stage analyzes the impact of the first four stages when looked at as a whole and presents a results analysis. Each stage consists of the following general actions where appropriate:

- Design recovery actions in preparation for additions and/or changes to any module.
- Additions and/or changes are implemented as needed to overcome the problem area being addressed.
- Performance measurements are taken and compiled through testing based on appropriate benchmark standards.
- The time and effort measurements for the stage are annotated.
- Added capabilities and/or increased tool utility introduced or affected by the changes made in a stage are annotated.

Stage one involves changing the programming environment under which OSMagic is being developed and then adjusting the current implementation of OSMagic to allow it to

work in that environment. The ObjectCenter C/C++ programming environment tool is now used for making and testing changes to OSMagic. There are initial settings that have to be learned and made to allow ObjectCenter to work with an ObjectStore application. The new tool needs to be learned and OSMagic requires some modifications to be compatible with this tool. For example, no DML can be used so any existing DML will have to be replaced with its equivalent C/C++ library interface representation. Responses to the work by Jacobs (8) included questions about the choice of the ObjectStore DML interface for OSMagic. Since that interface must be changed to support the use of ObjectCenter, an analysis of which interface to ObjectStore best suits the objectives of this work is also done in this stage.

Stage two implements those modifications required to make OSMagic a complete and fully functional conversion of Magic. This is accomplished by:

1. addressing OSMagic's inability to back up to a previous version of a design by determining the appropriate unit of *versioning* and then implementing it using ObjectStore's versioning mechanisms.
2. analyzing and changing OSMagic's data structures to insure they are properly allocated in either persistent or transient memory space.
3. identifying and implementing any other changes required to insure all of Magic's commands work in OSMagic.

Stage three removes the size limitation that OSMagic places on designs it can create or manipulate because of it's use of only one database transaction. This is accomplished by

1. determining the smallest unit of action possible in OSMagic so that the database remains consistent.
2. replacing the one database transaction with multiple transactions whose size is based on that unit of action.
3. analyzing the impact on concurrency control and failure recovery concerns introduced by the reduction of transaction sizes.

Stage four performs an analysis of how the capabilities of ObjectStore's versioning facility can be used to incorporate the cooperative design philosophy of the Magic users at AFIT. Alternative designs are presented for implementing such capabilities in OSMagic.

In stage five the changes from the previous four stages are considered as a whole and their impact on the tool analyzed. An analysis of the performance results gathered is presented and discussed as well important lessons learned in the conversion effort.

There is an expected overlap of some of these stages since the concepts each is addressing are not completely independent of each other. Every attempt is made to insure a coherent discussion of the issues this thesis addresses.

1.6 Materials and Equipment

For this research, we used version 1.2 of the ObjectStore object-oriented database management system. ObjectStore was developed by *Object Design Incorporated* of Burlington, Massachusetts. We also used version 1.1 of the ObjectCenter C/C++ programming environment tool. ObjectCenter was designed by *CenterLine Software, Inc.* of Cambridge, Massachusetts. The physical environment in which OSMagic resides was changed with the addition of a licensed NFS/NIS server added to the network of 22 Sun SPARC 2 workstations. In order to make ObjectStore available to multiple users, it was moved to physically reside on the new server named **Hawkeye**. This server has 128 MB of RAM and 256 MB of available swap space. OSMagic was moved to reside on the NFS server named **Cub**. To run OSMagic from the different Sun SPARC 2 workstations supported by this network, a client/server system was set up between the ObjectStore server on **Hawkeye** and the OSMagic application on **Cub**. This was done using the server, directory manager, and client manager parameter files accessed by ObjectStore on start-up. In this manner three Sun SPARC 2 workstations, **Maddie** and **Gepetto** both with 32MB of RAM and 64MB of swap space and **Nimrod** with 48MB of RAM and 96MB of swap space, were established as clients of the ObjectStore server.

1.7 Document Summary

Chapter 2 presents a review of pertinent literature in the areas of database transactions in the engineering environment and versioning models for cooperative design environments. It also reviews the facilities available in the ObjectStore DBMS to support these concepts. Chapter 3 details the design and implementation issues this effort undertook in

attempting to reach the stated goals. Chapter 4 then presents an analysis of the results this effort and Chapter 5 presents some final conclusions and recommendations.

II. Literature Review

2.1 Introduction

Understanding the design environment and how transaction and versioning mechanisms support that environment is essential if models are to be developed which will provide the means for implementing a fully functional OSMagic and the proper basis to extend OSMagic into a multi-user tool that supports a cooperative design environment. A thorough knowledge of ObjectStore's transaction and versioning facilities is required in order to design the transaction and versioning models which will best provide these capabilities in OSMagic.

2.2 Transactions in the Engineering Environment

A transaction, in the traditional sense, is an atomic unit of read and write operations against a database. They are atomic in that any changes made to persistent data within a transaction must not be made visible to other processes until the transaction successfully completes. Once a transaction does successfully complete, any changes it made to persistent data are committed to the database and made visible to other processes. If a transaction fails or is aborted for any reason, all changes made must be backed out of the database, returning the database to the state it was in when the transaction started. This means that while a transaction is accessing persistent data that data is locked and cannot be accessed by other transactions. In traditional business-oriented database systems where transactions typically only last a few seconds, this model is fine (3). This model, however, does not work in a computer-aided design environment like Magic.

Transactions in an engineering environment are different from those in environments supported by traditional database systems. In a typical engineering environment, complex designs are usually created and worked on by a team of engineers who work simultaneously on different pieces of the same design or on different designs that may share versions of the same component parts (13). During this *design* process engineers need to interact with each other in order to share design data and meta data (data about data). This environment is usually provided by a model consisting of a public database to manage

the design data in its stable state, and multiple private databases which interact with the public database. These private databases are usually on engineer workstations and are used to manage the design data in its unstable state, as its worked on by the engineers (11). A typical transaction on a private database entails the following (3):

- checking design data out from the public database into the private database,
- read and write operations on both the public and private databases,
- checking updated design data back into the public database.

Because of the nature of the *design* process, transactions can, and usually do, last a much longer time than transactions of a non-design application. It's not unusual for design transactions to last hours or even days. For this reason, transactions in a design application are called *long transactions* and those in traditional applications *short transactions* (13).

In traditional database systems short transactions are used both for concurrency control and recovery (11). Because of the short duration of such transactions, consistency is maintained by completely blocking access to data being used by a transaction until it has completed. Short transactions also serve as the unit of recovery since they change relatively few data items and thus can be rolled back easily. Due to their extended duration, long transactions cannot be used for recovery since many changes can be made and to roll back to the point when the transaction started could result in the loss of hours or days of work. Long transactions can still serve as a unit of consistency however if there are levels defined where changes are atomic (13).

Kim, *et al.* proposed a model for implementing *long transactions* which augments existing transaction models with the use of nested transactions and a modified database check-out environment (11). Their model adds the notion of a semi-public database to the environment described above. In this environment, designers can share less stable design objects (those not ready for the public database) by *downward committing* them to the less stable semi-public database. A transaction that checks out a data object from the semi-public database of another transaction becomes a child of that transaction. This results in a hierarchy of what are called nested transactions. A separate semi-public database is associated with each transaction in the nesting. A transaction can check data objects out

from the public database and/or any of the semi-public databases of its ancestors. This is called the check-out environment of the transaction. In this model, long transactions are defined as sequences of short transactions. Check-out and check-in requests are issued by each short transaction. All changes that short transactions make to public, semi-public, and private databases can be committed together. If a transaction aborts, roll back will occur to the end of the previous short transaction (11).

Bancilhon, *et al.* have proposed a model which expands on the model just described and corrects what they see as drawbacks in that model (3). Their model still uses the three types of databases, although differently named as global, project, and private. The idea of nested transactions is also maintained. The main difference is the view of the design environment. Bancilhon, *et al.* take into account the fact that some sort of windows environment is now likely to be run on a designer's workstation. This allows designers to create and manipulate multiple windows, each of which can be running multiple tasks concurrently. They say this defines the notion of a *set* of short transactions instead of a *sequence* as defined by Kim, *et al* (3). This model recognizes a finer breakdown of transaction types and a different *set* relationship between them. Nested transactions are described as a set of project transactions, each consisting of a set of cooperating transactions, each of which is a hierarchy of client transactions, each of which is a set of designer's transactions, each of which is a set of short transactions (3). A short transaction is started in a window of a designer's workstation and serves as the unit of consistency and recovery.

2.3 Versioning in CAD Applications

If a database management system (DBMS) is to support complex and data intensive design applications, it must support the concept of incremental and cooperative design. This concept is an inherent part of the design process and is the reason for the existence of multiple representations or versions of a design entity at any one point in time. For a DBMS to provide support for these kinds of applications, it must have some sort of version control capabilities. These capabilities are usually provided by a version management facility, which is an extension to the DBMS used to create and track information pertaining to the design of complex data entities.

Chou and Kim have proposed a model for versioning which explicitly incorporates the characteristics of CAD environments and the three types of databases (public, project, and private) which best support those environments (7). They model a CAD design by what they call a CAD object, which is represented as a configuration hierarchy. This hierarchy captures the relationships between the many component objects that fit together to make up the design. The authors define a set of operations for creating and manipulating designs and keeping track of the many versions each data object can have as well as which version objects are consistent with each other. Their system provides a complete history of a design object's evolution (7). With this model, designers can roll back to previous versions of a design and can branch off from any version of a design to explore new design paths.

Landis presents an approach to versioning that deals with the two basic kinds of evolution a CAD design experiences: linear and non-linear (14). With linear evolution, a design progresses in a linear fashion along a single design path. The current version of the design is the most recent. Only one view of the design is represented at any particular time. Designers can roll back to earlier versions of the design by directly referencing them. In the non-linear evolution case, a designer can create any number of branches from the current design. This produces new versions which can be used to explore alternative solution paths for the design. This also allows a team of designers to branch off and simultaneously work different parts of a design. In all cases two alternative versions of a design can be merged back into one on the main evolution path of that design. Landis defines numerous operations, which are used to create and manipulate versions, and a version hierarchy mechanism, used to manage the system (14).

The main action of versioning is on design objects. Version hierarchies are used to keep track of the derivation history of these objects. Configurations provide a means of grouping together all consistent versions of component objects which make up a composite object or a design. Alternatives provide a means for designers to explore different solution paths in a design and allows teams of designers to work simultaneously on the same design.

2.3.1 Design Objects. People, in particular design engineers, think in terms of abstract entities. In the design environment, these entities are best modeled as design objects

which encapsulate the identity of the object with the methods that define the objects' behavior. These objects become the building blocks of an engineer's design. Objects are the basic construct of the object-oriented data model and as such serve as the basic unit of versioning in database systems based on that model.

It has been the nature of design data that has molded the concept of objects and led to the development of the object-oriented paradigm. Data in design applications exhibit complexities not seen in applications handled by conventional database environments. Design data is dynamic in nature, can have multiple views, and multiple levels of abstraction (12). Design objects are often specified from different viewpoints and therefore can be associated with multiple representations of the design. A VLSI CAD design object, for example, can simultaneously have register transfer, Boolean logic, circuit, or layout representations (7). Multiple abstraction levels of design objects and their dynamic nature are a direct result of the phased approach to creating complex designs. This is the most common approach used and through its dynamic trial and error characteristic, design objects evolve throughout the design process by means of versions (12). Ahmed and Navathe see these complexities as the characteristics of design objects. They view design objects as "hierarchically formed assemblies of component objects" that give rise to multiple versions as they "go through the phases of design" (1). The relationships that defined these components at each level of abstraction form a hierarchy of the component objects which, when viewed as a whole, defines the entity being designed. A very simple example of this is shown in Figure 1. It shows a car door going through the phases of design and the versions that could result. Version two has added power windows and version three a power mirror. A door is a composite object which is composed of assemblies which are themselves are composite objects. Also the door will be part of a higher level of abstraction in the overall design of the car.

Since each component of a design can itself be a complex hierarchy of other components, it becomes clear that the designs' hierarchy is itself very complex. This means that the hierarchies used to manage versions can become even more complicated since any of the objects in these design hierarchies, whether primitive or composite, can have multiple versions associated with them.

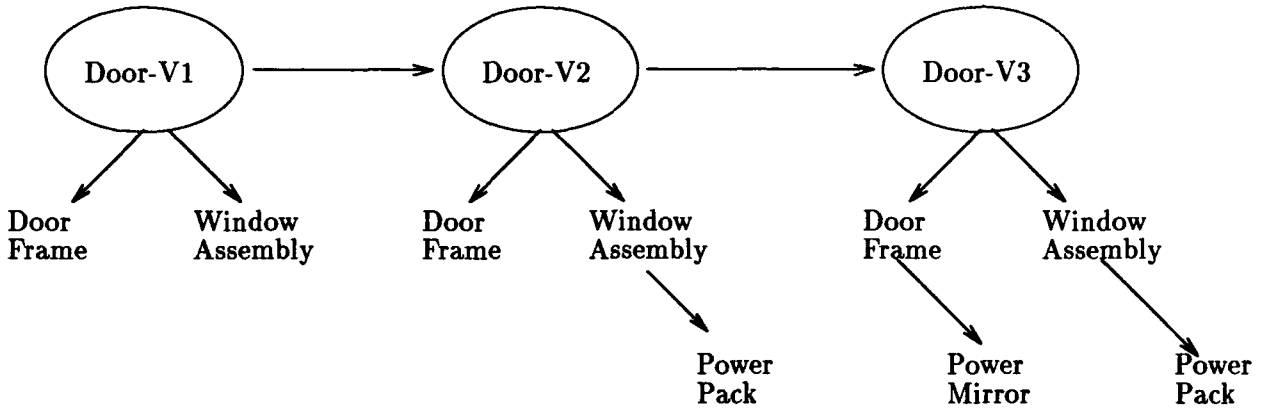


Figure 1. Door Design Object

2.3.2 Version Hierarchies. Version hierarchies are used to capture the derivation history (evolution) of versioned design objects. The most common representation used for these hierarchies is the version graph. Sometimes called history or derivation graphs, version graphs maintain the history of an object by means of *is-a-descendent-of* (successor) and *is-an-ancestor-of* (predecessor) relationships between the nodes of the graph. These relationships define a design object's evolution paths. The use of graphs allows for the capturing of both linear and non-linear evolution histories. In a non-linear evolution, a version object can have more than one predecessor or successor and therefore more than one evolution path (14). A version graph depicts all the versions in a version set. The ObjectStore database system uses what are called history graphs (15) and the ORION system uses the derivation graph concept (4) which is based on the work of Chou and Kim (7). In this approach, tables are used to represent the version derivation hierarchy. Figure 2 shows an example of a version graph. This graph depicts all versions of a particular cell object which are of the same generic type.

Version graphs are also used for tracking versions of configuration hierarchies (graphs). Configuration graphs keep track of which version objects from all the version sets involved in a configuration are consistent with each other.

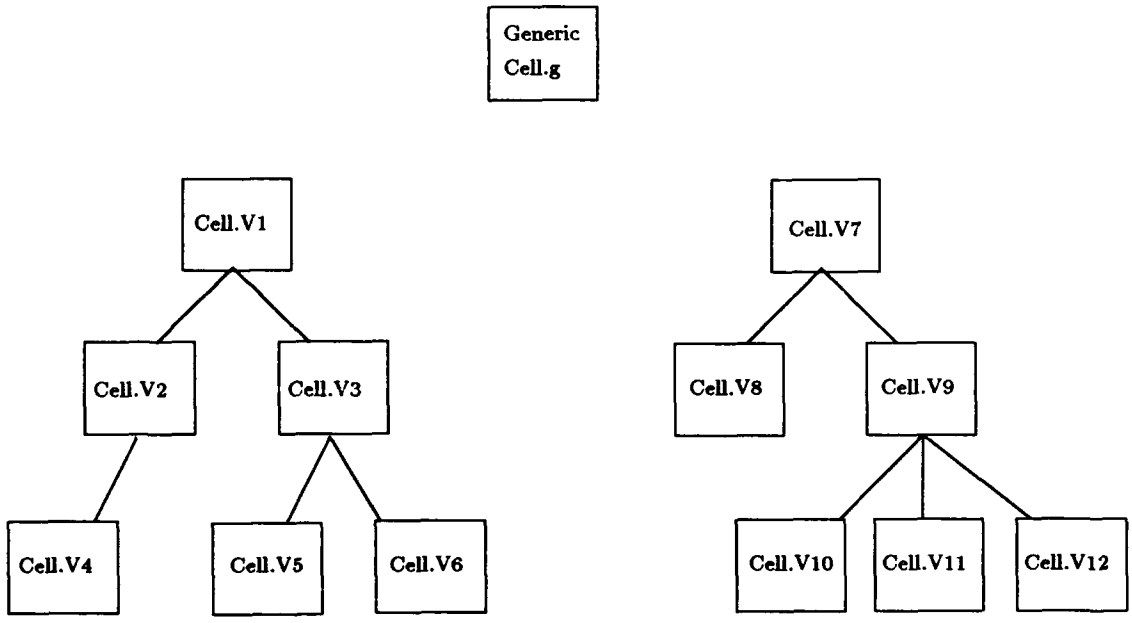


Figure 2. Version Graph

2.3.3 Configurations. A configuration is a means of grouping together all objects that are to be treated as a unit for the purpose of versioning (15). All related objects of the same version are grouped together in what is commonly called a configuration object. In other words, a configuration is a collection of the mutually consistent versions of all objects in a design (6). A configuration can be defined for any composite object. A design may be made up of a group of configurations or a configuration can represent a design. For example a VLSI chip is a configuration of cells which themselves are configurations of subcells which are configurations of still lower level component objects. Typically *is-a-component-of* or *is-composed-of* relationships are defined for configurations. The attributes of a configuration specify the components that belong to it. Configurations are related to each other by means of configuration hierarchies which are closely related to version hierarchies or derivation hierarchies which were discussed in the previous section. Configuration hierarchies are a good way of managing design data which are inherently hierarchical in nature (12). Most implementations and models consider configurations as objects, and as such they can be operated on like any other design object. Therefore all the versioning mechanisms for

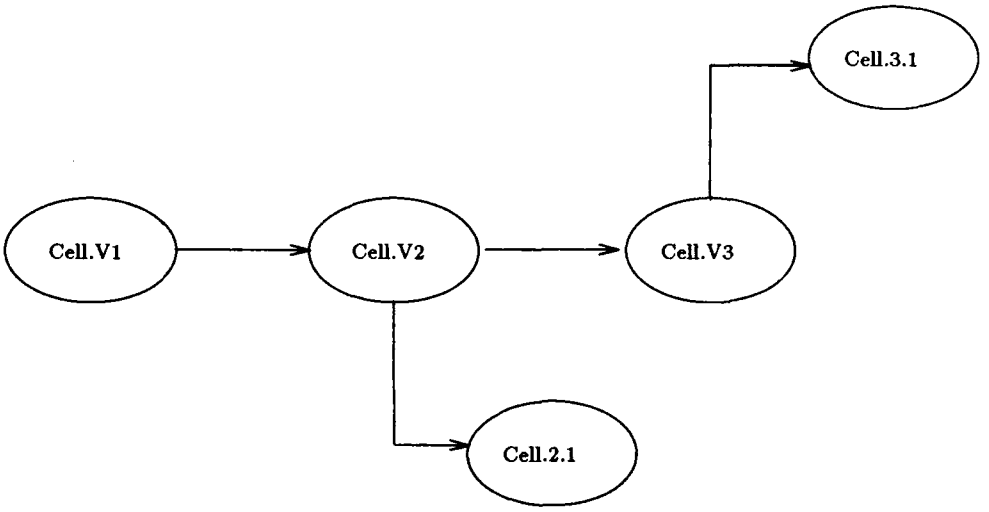


Figure 3. Alternative Versions

design objects apply to configuration objects. Several implementations on the market use configurations. The ONTOS database system by Ontologic uses configuration objects as part of its versioning mechanism (2) as does the ObjectStore database by Object Design (15).

2.3.4 Alternatives. Alternative versions are hypothetical variants of a transient or in-progress design version (9). They are different from other kinds of versions and are created as branches to the normal design path. In the linear evolution of a design, any changes we make result in a different version of the same solution path. Alternatives allow us to have multiple versions of multiple solution paths for the same design (Figure 3).

There are two main reasons designers use alternative versions. The first is to simultaneously pursue several different evolution paths of a design. This allows designers to study the trade-offs between several design options. There can be numerous scenarios in which this is the case. For example a designer may create an alternative version from the base or in-progress version of a design or design object in order to explore an experimental solution without affecting the in-progress version (9). In another example several engineers may be working on different solutions to the same design. Since in such a scenario designers cannot work on the same base version of the design, they would each check a copy of the

base or in-progress version into their own private workspace, thereby creating their own alternative versions to work with.

The second reason for using alternative versions is when a designer cannot mutate along a design's default evolution path because another designer has them locked out (14). This situation may arise when a team of designers are working on different parts of the same design. In this case instead of waiting, which in the design environment could be a long time, the designer can just branch off an alternative version and continue to work. Since in this scenario the designers are working concurrently, these versions are sometimes referred to as *concurrent* versions.

There may be a point when a designer wants to merge an alternative back in with the in-progress version in the default evolution path. Or a designer or team of designers may want to merge several alternative versions into one alternative version. This would be the case when a team of designers are ready to merge their parts of a design in order to create one new version of the design. To do this some sort of merge facility is needed. Most versioning models call for such a capability (5, 9, 14) and some commercial OODBMSs have implemented some sort of merge facility. For a merge facility to work, the actual resolving of conflicts and choosing of desired features must be determined by the designers involved. The ObjectStore OODBMS by Object Design allows for the merging of versions by creating a new version which is a successor of each of the versions that are to be merged. The user then modifies this version taking what is desired from each of the other versions. The user then checks the new merged version into the appropriate workspace so that it becomes the new working version that is resolved to by the parent workspace (See Figure 4).

2.4 Versioning Issues for Cooperative Work Models

Versioning plays an important role in the conversion of CAD applications to the use of a database management system. Such a conversion not only involves the direct changes to code required to store and retrieve data from the database, but also the changes needed to incorporate the cooperative work environment philosophy which best suits the application. This requires an understanding of the design environment the application supports, the

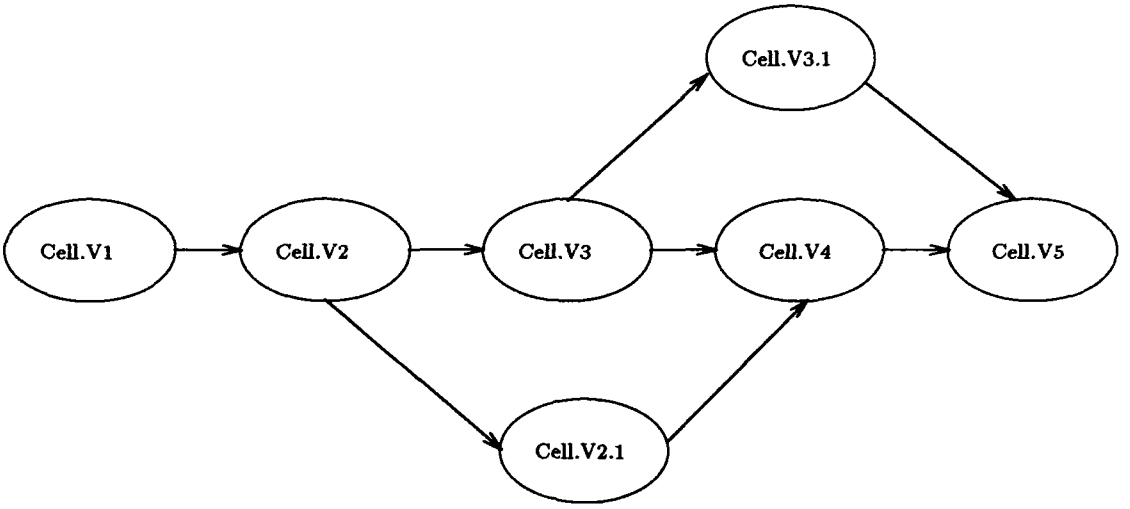


Figure 4. Merging Alternative Versions

idea of a workspace model, and the concept of version states. Each of these concepts must be folded into any conversion of a CAD application to a database implementation.

2.4.1 Design Environment. Versioning is an inherent part of the design process and thus the design environment. The design environment is the conceptual and physical environment in which designs of a particular nature reside (12). The design process is characterized by the conscious evolution of data towards some defined goal (14). In realizing this goal designers will use a variety of options available to them like the configurations and alternatives discussed in the previous section. The physical environment in which the design process takes place usually consists of several levels of databases or workspaces. All such environments discussed in the literature reflect the concepts that are defined in what is known as the *workspace model.* The conceptual levels of a design object are defined by the states the versions of that object go through as it evolves through the various physical workspaces.

2.4.2 Workspace Model. Examining the workspace model defined for *Version Server*, the prototype OODBMS used in the work of Katz and Chang, provides a good overview of the physical side of the design environment (10). In this model, designers *check-out*

data objects from SHARED ARCHIVES into PRIVATE WORKSPACES. They make any design changes in this private workspace. These changes cannot be seen by anyone who doesn't have access to that workspace. When and if it is desired, the changed objects are *checked-in* to what is called a SHARED GROUP WORKSPACE. In this area the modified objects can be integrated with the work of other designers. When all changes to an object are made and validated, the object is checked in to the SHARED ARCHIVE as a new version. Just about all design environments use some form of the workspace model. Chou and Kim define public, private, and project databases which correspond to the shared, private, and shared group workspaces just described (7). They describe this as a client-server environment, where clients are the private databases and a central server controls the public database while intermediate servers control the project databases. This client-server model works the same as the workspace model. The ObjectStore OODBMS implementation uses the workspace model (15). It should be noted that not all environments use projects in which case there are only two workspaces or databases. Also, some designs are not worked as projects and would therefore bypass the project workspace. Figure 5 depicts a workspace environment and the relationships that can occur between workspaces. The lines indicate the *check-in* and *check-out* relationships that exist. Note that design objects cannot be checked out of private workspaces. The objects in these workspaces are completely controlled by the owner of that workspace.

In this type of physical environment, as objects are moved between the different workspaces, new versions are created in order to maintain consistency between data objects and to provide a mechanism for failure recovery. Versions then, model the *movement* between workspaces and this *movement* defines the different states versions take on as they move through the design process.

2.4.3 Version States. Versions will take on different states depending on where they are at in the design process. Chou and Kim define three states (they call them version types) that design versions can be in (7). These are *released, working,* and *transient.* They define the properties and constraints a version will have in each state as follows:

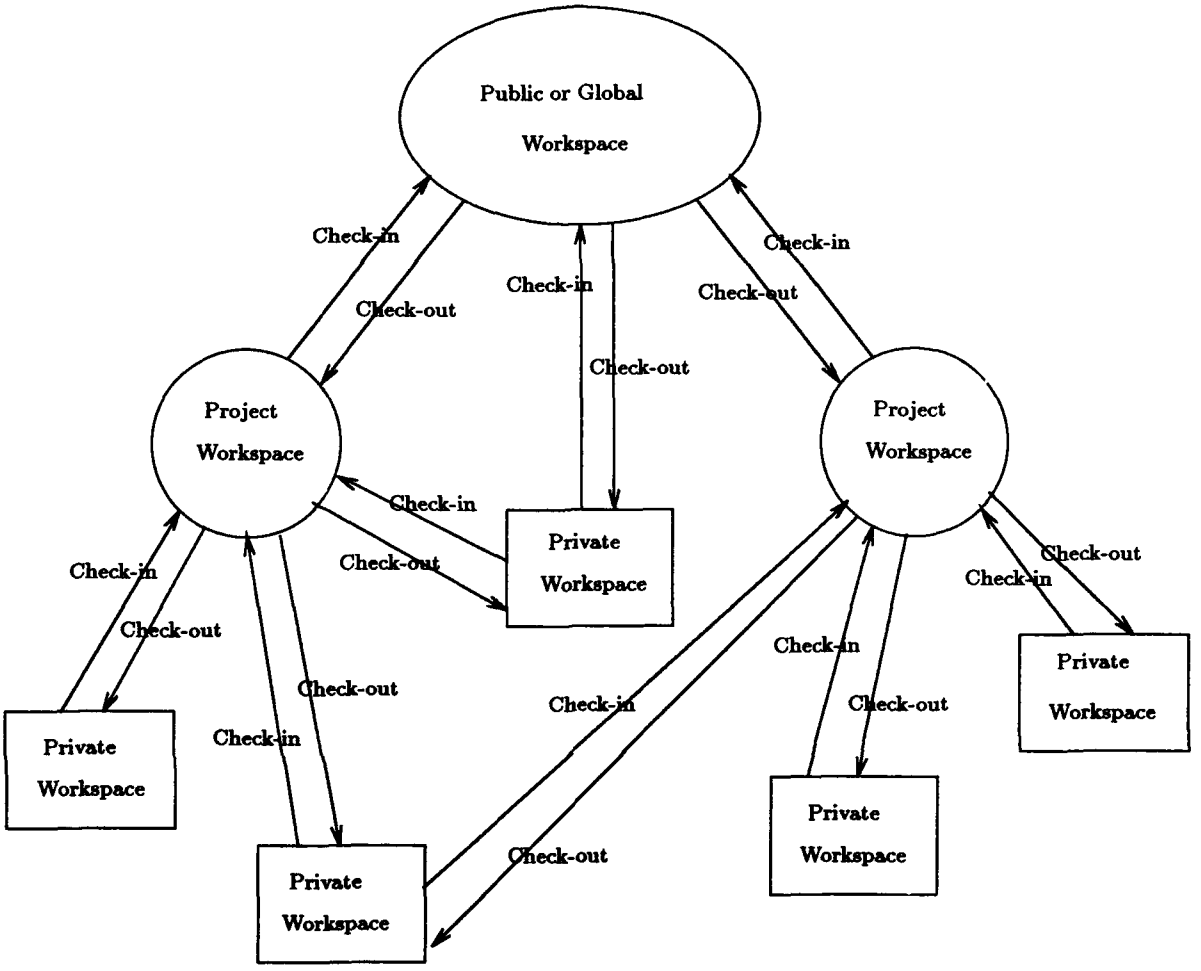


Figure 5. Workspace Model

- Properties of a *released version*:
  1. It resides in the public database, and is managed by the database server.
  2. It cannot be updated.
  3. It cannot be deleted.
  4. A transient version can be derived from a released version.
  5. A working version can be promoted to a released version.

- Properties of a *working version*:
  1. It can exist in a private database or in a project database.
  2. It is considered stable and cannot be updated.
  3. It can be deleted by its owner.
  4. A transient version can be derived from a working version.
  5. A transient version can be promoted to a working version either explicitly or implicitly.

- Properties of a *transient version*:
  1. It can be updated by the designer who created it.
  2. It can be deleted by the designer who created it.
  3. A new transient version can be derived from an existing transient version. The existing transient version is then promoted to a working version.
  4. It is stored in the private database of a designer who created it.

It is clear from these definitions that the state of a version is closely related to the workspace in which it was created.

With this understanding we can explain the relationships that occur between the version states defined by Chou and Kim (7) and then show how others approaches are similar or different. These relationships basically define the creation of versions. When a designer wants to work on a design object, the version of that object is checked out of the workspace (database) in which it resides into the designer's private workspace (database). The effect of this operation is the creation of a new transient version of the object in the designer's workspace which is a copy of the desired version. The version being copied is not affected. Any work the designer does with this version is not seen by, nor does it affect, the other workspaces unless it is explicitly checked into one of those workspaces by the designer who owns it. A new transient version can be created by checking out a released version from a public workspace, checking out a working version from a project workspace, or deriving a new version from an existing transient version in a designer's private workspace. Notice there is no concept of checking out a version from a private workspace. A designer's version of a design object remains private until he/she makes it available to other designer's by checking it in to either the project or the public workspace.

If a new transient version is derived from a current transient version, the current transient version is automatically promoted to a working version in the designer's workspace.

There are two ways a designer can make their version of an object visible outside their private workspace. One is to check a transient version into the project database so it can be seen by other designers in the project and the project administrator. This operation causes a new working version of the transient version to be created in the project workspace. This new working version becomes the current working version of that object in the project workspace. The previous working version and the transient version remain unchanged. The other way is to check a working version of the object from their private workspace into the public workspace. This operation causes a new released version of the private workspace working version to be created in the public workspace. The new released version becomes the current released version of the object. The previous released and working versions remain unchanged. This situation would arise if there were no project workspace in the application or the design object being manipulated was not part of a project effort. If there is a project workspace, then another way to create a new released version of an object is to check a working version in from the project database. The affect is the same as that just described above.

When a designer wants to create a more stable version of an object within their private workspace, they would promote the transient version of the object to a working version. As mentioned before, a working version will be automatically created in the designer's private workspace if they derive a transient version from another transient version. A working version in a *private* workspace is owned by the designer who created and promoted the transient version. A working version in the *project* workspace is owned by the project administrator.

Changes can only be made to the transient version of an object. Obviously, changes to a released version cannot be allowed. There are several reasons why working versions cannot be updated. First, working versions are considered stable. This means transient versions can be made from them and if the working version were to be changed, it would have to be in such a way as to prevent those transient versions from seeing the changes. But this can be done by just creating another transient version and using it to update

the working version. So there is no need to create any special operations for such changes and the "no update" constraint can be placed on working versions. Second, by having this constraint there is no need to support a separate update mode check-out mechanism. Instead, all check-outs can remain read-only. This means that transactions issuing check-out requests do not have to be blocked.

In Chou and Kim's model there is no limit the number of working versions that can be defined on a design object. This allows a designer to check two transient versions, which were derived from the same working version, into the project database as two different working versions. Other designers in the project can then work with these different alternatives of the object.

The previous discussion was based mainly on the model proposed by Chou and Kim (7). This approach was implemented in the ORION database system (4). In that implementation, their *released version* state was not used. Others that discuss version states or types present similar approaches. For example, Ahmed and Navathe define the possible states of a version as validated, stable, and transient (1). While they do not discuss the environment this would work under, it isn't hard to see how these states match up with those presented above. Validated versions are the same as released versions, stable the same as working, and transient the same as transient except that other transients cannot be derived from transient versions. In this approach, like the other, all new versions begin in the transient state, but they do so by being created from generic versions or by being derived from validated or stable versions. This means that an implicit promotion of transient version to the stable state is not required. All promotions are explicit in nature. Figure 6 shows the relationships between version states in this model (1).

This concept of version states is even seen in the early work of Katz and Lehman (9). In their proposal they refer to in-progress (transient), effective (working, stable), and released (validated) version states. These version states are manipulated as design files in a similar manner to the workspace model described above.

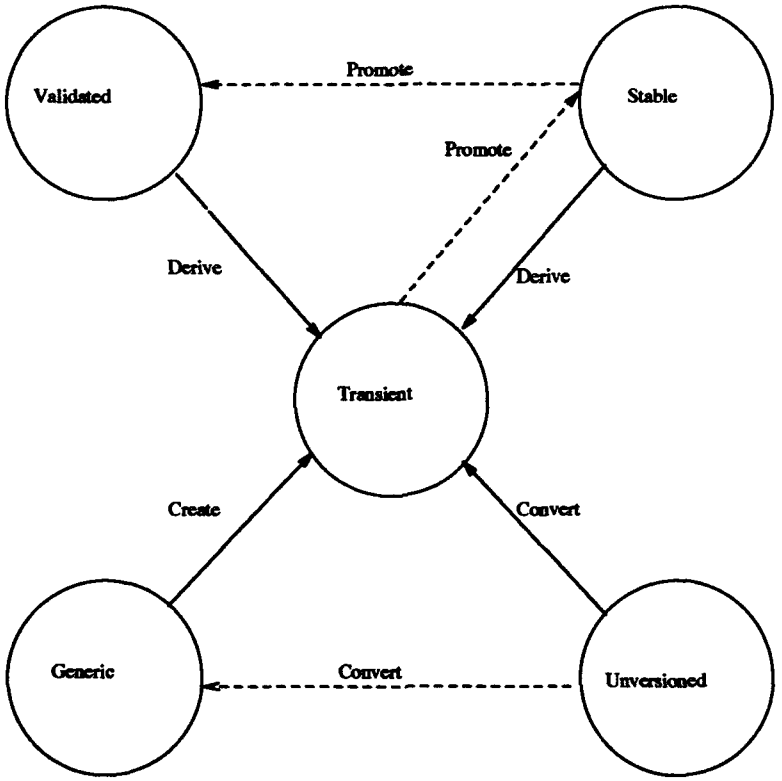


Figure 6. Transition of Versions States

2.5 The Transaction and Versioning Facilities of ObjectStore

ObjectStore's transaction and versioning facilities both play an important role in the implementation of a design application with the database. The transaction facility is typical of any database, providing for atomic operations on persistent data which guarantee data consistency and concurrency control when properly implemented. The concepts of hiding intermediate results of operations and rolling back persistent state are supported by the ability to nest and abort transactions. These and other features of what are known as *short* transactions are supported. The concept of *long* transactions and the support for cooperative work environments are provided by ObjectStore's versioning facility. This facility defines configuration and workspace models and provides operations for implementing both. The configuration model allows for defining configurations of design objects at multiple levels of granularity. The workspace model supports the definition of a hierarchy of workspaces with more restrictive access as you work down through the levels. This supports the concept of workspaces for global libraries, project work areas, and the more

restrictive individual designer workspace. By checking configurations in and out of the various workspaces the concept of *long* transactions is supported via locking mechanisms for the configurations. The objects defined by a configuration are locked when checked-out and remain so until checked back in to the parent workspace. The concept of cooperative design is supported by the ability to create alternative versions of design objects and eventually merge those objects back together in the version hierarchy. To provide the proper visibility to the different versions of a design object, ObjectStore automatically *floats* the pointers to those objects to the appropriate version. This is accomplished by ObjectStore's memory mapping architecture when versioned objects are mapped into virtual memory (15).

2.6 Summary

The transaction and versioning models presented here define basic frameworks and guidelines to work from when incorporating transaction and version management mechanisms in computer-aided design applications. Each of these models, transaction and versioning, recognize the nature of the design environment. The three types of databases that the transaction models deal with define the same database system the OODBMS models are serving. Both concepts address the consistency and recovery capabilities required by complex CAD applications. It is inevitable that the implementations of these two concepts will work hand in hand to provide the kind of database support that is required by systems like Magic. If OSMagic is going to provide the full functionality of Magic, it will have to incorporate a proper implementation of the transaction and versioning mechanisms provide by the ObjectStore OODBMS.

III. Design and Implementation

3.1 Overview

As with the first effort, this effort to make OSMagic a fully functional implementation of Magic using ObjectStore requires the sound use of recognized software engineering principles. In each stage of this work, problem areas and/or desired enhancements are identified and an analysis conducted of the existing design to understand the causes of the problems or the impact of the enhancements. Then solutions are designed and implemented based on that analysis, and testing conducted to insure the fix and or enhancement was implemented satisfactorily and meets desired performance goals. In this chapter we detail the work in the first four stages of this effort. We discuss the goal of each stage, the approach we took in trying to achieve that goal, the designs we implemented, and the problems we encountered. We also highlight what we feel are some important lessons learned which will apply to any conversion effort of this type.

3.2 Environment and Interface Issues - Stage 1

The goals of this stage were to provide a better environment in which to further the development of OSMagic and to address the concerns raised about which interface to ObjectStore would better serve the objectives of this research. Changes in the physical environment, the movement of the ObjectStore software, and the desire to use the ObjectCenter C/C++ programming environment tool for making modifications to OSMagic, prompted this effort. The use of ObjectCenter would provide the advantages of a programming and debugging environment that makes the analysis, modification, and testing of OSMagic much easier and more organized. Questions raised about why the first OSMagic used the ObjectStore DML interface and the need to change that interface to use ObjectCenter provided the reasons to address the interface issue.

3.2.1 OSMagic's Interface to ObjectStore. ObjectStore provides three interfaces applications can use when connecting with the database: the C++ Library Interface, the C Library Interface, and the DML Interface. Since ObjectStore is a C++ based application, the C++ interface is more general and better documented. The C interface has

corresponding instructions for most but not all of the C++ interface but is less straightforward to implement. The DML interface is used in conjunction with the C++ interface and provides more convenient versions for some of the C++ interface instructions. The DML interface does many things which make it easier to use, like the automatic definition of database roots for persistent variables. However, because the DML instructions make use of features which are specific to ObjectStore's C++ compiler, an application which uses the interface is not compatible with tools using other compilers.

There are two main reasons this thesis addresses the issue of the interface to ObjectStore. First is the desire to use the ObjectCenter programming tool for continued work on OSMagic. ObjectCenter does not recognize ObjectStore's DML since it uses the more general AT&T C++ compiler, version 2.1. To use ObjectCenter to further develop OSMagic requires changing from the ObjectStore DML interface to the more general C++ interface, which supports any AT&T cfront compiler, or to the C interface. Secondly, concern was expressed in responses to the work by Jacobs (8) as to why OSMagic used ObjectStore's DML interface as opposed to its C Library interface. These responses questioned whether or not ObjectStore's C interface would have been a better choice since Magic is written in C. Since the interface was to be changed anyway in order to use ObjectCenter, it was felt that it wouldn't require much more effort to code the OSMagic application so it could be compiled and linked using any one of the three ObjectStore interfaces. This would allow for the use of ObjectCenter, provide a means of gaining insight into the interface question, and allow for continued use of the database created in the initial OSMagic effort.

This effort was not as straightforward as anticipated, however. In ObjectStore, database roots must be defined in order to access data. These roots provide access points into the database which are then used to traverse to any desired data objects. Since we want OSMagic to access the same database when set up in any of the three interfaces, the same database roots have to be accessible via any of the interfaces. This proved to be a difficult task. ObjectStore's DML adds an extra level of indirection when defining database roots and persistent variables. By working with the Object Design support branch, we determined the proper method of defining roots in the three interfaces such that OSMagic could access data in the same database via the same database roots. We were then able

to implement all ObjectStore instructions used in OSMagic in all three ObjectStore interfaces. We did this by defining preprocessor variables for each interface and wrapping the appropriate version of each ObjectStore instruction in preprocessor statements controlled by these variables. The interface desired for a particular run can then be used by activating the corresponding preprocessor variable with the appropriate compiler option. For details on exactly what changes were required and examples of how they were implemented see Appendix A.

3.2.2 Using ObjectCenter. ObjectCenter is a C/C++ programming tool which provides a complete environment for editing, compiling, linking, running, and debugging C and C++ applications. It is an interactive tool which also provides graphical browsers which depict the relationships between data objects and aid in program understanding. The use of this tool would provide a much better environment for the continued effort on OSMagic. As discussed in the previous section, to use this tool for OSMagic required changing the interface to ObjectStore. We also discovered that to use ObjectCenter for an ObjectStore application required special makefile targets and compilation considerations. Again these changes were not as easy as first thought. Also, we were unable to determine if ObjectCenter could access all of OSMagic's source code when needed because of the way in which OSMagic is laid out. OSMagic's source is kept in 36 modules. Each module has a Makefile which compiles the files located there and links them together into an object file for the module. The top level Makefile then links all the module objects to create the OSMagic executable. Ultimately we gave up on using ObjectCenter for OSMagic due to time constraints and the unresolved issues. Those issues we were able to solve are discussed in detail in Appendix B.

3.3 Making OSMagic Fully Functional - Stage 2

To be able to make conclusions about the feasibility of converting existing design tools so they can have the advantages of a database, one must first and foremost be able to show that the *new* tool retains all the capabilities of the *old.* Then and only then can the issues of added capabilities, performance comparisons, the conversion effort, and the

tradeoffs between them be discussed with confidence that all relevant factors are being addressed. The goal of this stage therefore is to pick up where the first effort left off by first insuring OSMagic is a fully functional implementation of the original Magic.

Based on the results of the initial effort (8) we thought the only functionality lacking in OSMagic was the ability for designers to roll back to previous baseline versions of a design. For this reason we started our work by picking up where the first effort left off in the implementation of ObjectStore's versioning facility. The proper implementation of versioning will not only give OSMagic the ability to roll back to previous versions but also provide the underlying framework needed for OSMagic to take advantage of the added capabilities it will have once properly interfaced with the ObjectStore database.

However, while implementing the first phase of the versioning model, two major problems surfaced that, when analyzed, showed that many of the commands available in the original Magic would not work in OSMagic. Temporary work objects which should be transient were persistently allocated and illegal transient addresses were discovered in the database. These problems were discovered at different times during the implementation of the first phase of the versioning model and while not directly related to that effort had to be solved before proceeding.

3.3.1 Persistent Declarations of Internal Work Objects. While testing the implementation of our initial versioning model, a fatal error was encountered involving an attempt to access a frozen version of a data object. Analysis found that the problem was caused by the fact that the initial OSMagic prototype persistently allocated many temporary cell definitions, used internally by Magic in the implementation of its algorithms, when they should have remained transient. The surfacing of this problem shed light on the fact that there is a high probability that an existing CAD tool will have data of the same type that must be both persistently and transiently allocated and manipulated if the tool is converted to a database implementation. The existence of such data types impacts the data allocation functions which create and initialize the data structures used by the tool and possibly the algorithms which implement the tool's commands. A thorough analysis

will be required by any such conversion to determine which data types fit this category and what changes are required to insure the application retains full functionality.

3.3.1.1 Summary of Problem Analysis. Magic defines and uses many temporary work cells in its algorithms for implementing its available commands. These cells are internal to Magic and only last for the life of the process. They are never written or saved to disk. The first OSMagic made all cell definitions persistent, including the *internal* work cells. When we implemented our initial versioning model in this effort, it made configurations of each persistently defined cell definition and thus of the internal cell definitions as well. When new configurations are created and loaded into the database, they are initially checked-in to the global workspace and are only checked-out by *read* operations. Since internal work cell definitions are never explicitly read by the user, they were checked-in the global workspace when created but never checked-out. This caused the *attempted access of frozen version* errors. The problem didn't surface sooner since the versioning code had not been implemented. Also, since all of Magic's commands were not tested in the first effort, the problems the internal cells would have encountered weren't discovered.

The analysis of how Magic uses its internal cell definitions made it clear that they should not be persistently allocated in OSMagic, but rather transiently allocated as in Magic. This will insure they go away at the end of the process and are recreated each time the tool is used. There is no reason for saving these cells in the database since the information they hold is of no use. To do so without special provisions for reinitializing them would only introduce errors in the tool's algorithms.

3.3.1.2 Summary of Problem Solution. There are three classes of cell definitions OSMagic must address. First are the temporary work cells used only in the internal work algorithms of the tool. Because the data in these cells is never saved, they do not require persistence. They should be allocated in transient memory as they were in the original Magic. Second are the cell definitions which represent the designs and definitions created and manipulated in OSMagic and saved in the database. These cell definitions must be allocated in persistent memory and configured for versioning purposes. They must be configured so their design versions can be tracked and used for the implementation of

Magic commands and for control of the added multi-user capabilities the database brings to the tool. Third are the temporary cell definitions used only in internal algorithms but which may contain data which is to be saved in persistent memory space. Because they can contain persistent data, they must be considered as a separate class to insure appropriate actions are taken to handle that data. In OSMagic there are two ways to handle this class of cell definition. One approach is to allocate them in persistent memory space so that any data objects created in them are persistent. They should not be configured for versioning purposes however, since such cells are never stored in the database. Instead these cell definitions should be reinitialized or recreated at appropriate times. This approach allows the commands to work essentially the same as in the original Magic. Because we initially felt that our versioning model must support the algorithms for the commands of OSMagic, this is the approach we used. We found, however, that it wasn't the best approach to use since the persistent allocation of temporary data structures deviates from natural logic and is less efficient since it requires database access for each reference to those structures. It also requires extra clean-up code to remove these cells from persistent memory when exiting the tool.

A better approach is to treat the cells of this class as any other internal cell definition. In this approach such cell definitions are allocated in transient memory and the commands that assign any data in them to persistent space are appropriately modify. This approach is superior to the first since it defines a more efficient and natural implementation of this class of cells. While new code is needed to insure proper allocation of space when saving the data in these cells to persistent space, there is no requirement for clean-up code as in the first approach. Unfortunately, because of the time limitations we faced we had to stay with our implementation of the first approach. However, we did learn a valuable lesson we feel will apply in any conversion of this type. The models and solutions necessary for a conversion do not have to be designed around the algorithms of the existing tool. The algorithms can be changed to fit the models or solutions. The approach used should be that which provides the best solution to the problem at hand while maintaining the overall functionality of the tool. Appendix C contains a detailed discussion of the analysis of this problem and the changes required to overcome it.

3.3.2 Transient Addresses in Database Once the problem with transient and persistent cell definitions was solved, another problem became evident. ObjectStore was finding transient addresses in the database. ObjectStore defines such addresses as illegal, which when found will cause fatal errors. This problem was a show stopper since it meant our database was inconsistent. Until all cases of these illegal addresses were identified, the reasons for their improper allocation determined, and solutions implemented we could not continue with the conversion effort.

Analysis found that some component fields of OSMagic's persistent hash tables and cell definitions were **not** allocated in persistent memory when created and initialized in the initial prototype. Therefore transient, and thus illegal, addresses were stored in the database for these fields. Having ascertained why the database was bad, the next step was to identify all areas in the database where transient addresses appeared and implement the necessary fixes. Illegal pointers can be found by using ObjectStore's `database::set_check_illegal_pointers(1)` option. Setting this option to **1** forces ObjectStore to check for illegal pointers in all segments of the database. The default setting of this option is **0** which gives ObjectStore the option of making the checks or optimizing them out. By explicitly setting this option to **1** and using the ObjectStore database browser, all components in the database with transient addresses were identified. The problem occurred in the *ti_client,* and *ti_body* fields of the **Tile** data structure; the *cu_client* field of the **CellUse** data structure; the unassigned pointers of the **osHashTable** data structure; and the *infinityTile* which is assigned to certain tiles of the **Plane** data structure.

While some of these fields were quite easy to fix, others were not. Because of its very detailed nature, the discussion of the solution analysis and implementation for each of the identified fields has been placed in Appendix D. One thing that stands out in that discussion and is important to note here, is how the programming techniques used in the original Magic impacted the effort required to implement the solutions to these allocation problems. This is especially true for those techniques and shortcuts which take advantage of language-specific side effects. A prime example of this in Magic is the use of *casting* to store data structures of different types in the same area of memory instead of using the `UNION` construct defined for that purpose. Such techniques cannot be used when allocating

persistent memory space. The fixes for the OSMagic fields affected by this technique were much more complicated than they would have been if proper structured programming had been used in the original tool. Programming techniques and shortcuts like this will probably be encountered in any conversion effort of this type. Their impact can be greatly reduced by early identification.

3.3.3 Versioning Issues. In order for OSMagic to have the same functionality of the original Magic, a means of backing up to a previous version of a design object is required as well as a means to save the current version being manipulated before continuing work. The **flush** and **save** commands require these capabilities. The best way to implement these commands using the capabilities of ObjectStore and at the same time lay the foundation required to properly handle the added multi-user capabilities of OSMagic is to incorporate ObjectStore's versioning mechanism. This section discusses the initial versioning model which was needed to implement these commands in OSMagic. The more sophisticated model needed for implementation of an OSMagic which has multi-user capabilities is discussed later.

The two constructs that make up the foundation of ObjectStore's versioning are *configurations* and *workspaces*. Configurations define how different design objects are grouped together for versioning purposes. Workspaces are used to create and manipulate versions of configurations and determine visibility to the various states of those versions. The implementation of a versioning model requires definition of the configurations needed by the application and then definition of the workspaces where those configurations will reside and be worked upon.

3.3.3.1 Configurations. ObjectStore implements and controls versions of design objects by means of a data class called a **configuration**. A configuration defines a *version* object which is made up of all the data objects which are to be considered as a whole for versioning purposes. By properly defining what constitutes a configuration in the application, and implementing the proper movement of those configurations through the various workspaces, the application's versioning model is implemented.

In order to implement the ObjectStore versioning facility in OSMagic, we first had to decide what should make up a configuration. Analysis showed that the main unit of work in the OSMagic tool is the cell. There are two main data structures used for defining and manipulating cells in OSMagic, the *cell definition* and the *cell use*. There is one cell definition per cell, which is made up of numerous smaller data structures which together completely define the cell. There are multiple cell *uses*, one for each instantiation of the cell. All of a cell's *uses* are kept in a list which is also a part of the cell definition. This makes the cell definition the main data structure in OSMagic. As such, it appears to be the best choice for the definition of a configuration. By configuring the cell definition and all of its parts, we insure proper versioning by having grouped all the data parts that should evolve together in one *version* object.

A cell definition also includes a list of all of the instantiations of all subcells used by that definition. These subcells are themselves defined by cell definitions which along with all their uses and parts make up their own configurations. The question that arises here is whether or not these subcells should be made subconfigurations of what is considered the main cell definition. To answer this question, and certainly before trying such an implementation, a complete understanding of how OSMagic uses subcells is needed. If the goal is to maintain a library of basic cells which can then be made part of larger designs, then making them subconfigurations when they already exist as configurations on their own may not be possible. Also, if the tool will by its nature insure the proper checking in or out of all cell definitions used together in a design, then subconfigurations are not required. Another question to investigate is whether a subconfiguration can be made part of another configuration. The answers to these questions will require a deep analysis of how OSMagic implements its subcells, the relationships involved, and the goal of its eventual multi-user implementation. For this first phase implementation, all cells, whether eventually a subcell of larger definitions or not, are made configurations in their own right.

The implementation of configurations requires defining a new configuration for each new cell definition and maintaining a hash table of them as is done for cell definitions. Then whenever a new component object of a persistent cell definition is allocated, it

```
if (configure_it) {

  cellConfig = new (magicdb1, 1, CellConfig_type) configuration();
  configEntry = osHashFind(dbCellConfigTable, cellName);
  osHashSetValue(configEntry, (void*) cellConfig);
}

if (objectstore::is_persistent(cellDef))
  if ((cellConfig = configuration::of(cellDef)) != 0)
    cellUse = new(magicdb1, cellConfig,  1, CellUse_type) CellUse;
  else
    cellUse = new(magicdb1, 1, CellUse_type) CellUse;
```

Figure 7. OSMagic Configuration Declarations

is allocated in the configuration of that cell definition, if one exists. The ObjectStore command `configuration::of` is used to determine whether a cell definition is configured or not and if so returns a pointer to that configuration which can then be used to allocate the new component in it. Configurations are created in ObjectStore with the `new () configuration` command. Figure 7 shows the creation of a new configuration and an example allocation of an object in that configuration using the C++ Library Interface. New configurations are created in the current workspace and then checked into the global workspace. To access objects defined by a configuration, that configuration must have first been checked out of the global workspace into the current workspace. This then required the proper setting of workspaces followed by code which insures needed configurations are checked out at the proper time.

3.3.3.2 Workspaces. Only two workspaces are needed to implement OSMagic as a single-user version of Magic implemented on ObjectStore. One to act as the global storage area for the configurations being manipulated and one for the user work area. With these two workspaces, a model can be implemented which allows a user to work on a design object by checking its configuration out of the *global* workspace into the *user* workspace. With ObjectStore, this check-out operation creates a new version of the configuration in the *user* workspace and freezes that configuration until it is checked back in by the user. This simple model, in conjunction with a proper transaction model, allows for the implementation of a fully functional OSMagic.

```
OS_BEGIN_TXN (tx0, 0, transaction::update) {

   database_root* ws_root1 = magicdb1->find_root("REALglobal_ws");
   if (!ws_root1) {
     global_ws = workspace::create_global(magicdb1);
     database_root* ws_root1 = magicdb1->create_root("REALglobal_ws");
     ws_root1->set_value(global_ws, Workspace_type);
   }
   else
     global_ws = (workspace*) ws_root1->get_value(Workspace_type);

   database_root* ws_root2 = magicdb1->find_root("REALuser_ws");
   if (!ws_root2) {
     user_ws = new (magicdb1, Workspace_type)
                          workspace(global_ws, "user_workspace");
     database_root* ws_root2 = magicdb1->create_root("REALuser_ws");
     ws_root2->set_value(user_ws, Workspace_type);
   }
   else
     user_ws = (workspace*) ws_root2->get_value(Workspace_type);

   workspace::set_current (user_ws);
}
OS_END_TXN (tx0)
```

Figure 8. OSMagic Workspace Declarations

In ObjectStore, workspaces form a parent-child hierarchy where versions of design objects are checked out of the parent into the child. This is implemented in ObjectStore by first defining a global workspace with the **workspace::create_global** command. Then all other workspaces are defined by using the **new() workspace** command in which the new workspace's parent can be specified. Then in the application, a current workspace is established with the **workspace::set_current** command. Whenever a check-out or check-in operation is performed on a configuration the parent of the *current* workspace is used. Figure 8 shows how we defined two workspaces, the **global_workspace** and the **user_workspace** in our initial versioning model implementation.

With this simple workspace implementation, the proper defining of configurations, and the proper transaction model, we can implement all of the original Magic commands in OSMagic. For example, the **flush** command, which was not available in the first OSMagic, allows a user of Magic to forget all work done on the current version of a design object and

```
if ((cellConfig = DBCellLookConfig(cellDef->cd_name)) !=
                                          (CellConfig*) NULL)
{
  OS_BEGIN_TXN(tx2, 0, transaction::update)
  {
     cellConfig->forget();
     cellConfig->successor()->destroy_version();
  }
  OS_END_TXN(tx2)

  OS_BEGIN_TXN(tx3, 0, transaction::update)
  {
     cellConfig->checkout();
  }
  OS_END_TXN(tx3)
  return TRUE;
}
/* if it doesn't exist, print error */
else
{
  TxPrintf("Cell configuration doesn't exist.\n");
  return FALSE;
}
```

Figure 9. Implementation of Flush Command

go back to a previous version. Magic does this by first deleting the design's internal cell definition and then reloading the design from disk. Thus the design object's cell definition is redefined based on the last version saved. Our basic versioning model allows us to implement this command in OSMagic by first using the ObjectStore `configuration::forget` command to forget the current version of the design object, the cell definition in this case, and then using a combination of the `configuration::successor` and `configuration::destroy` commands to remove the forgotten version from the version hierarchy of the parent workspace. This allows us to check a new version of the design object out of the parent workspace which is the same as the version that was flushed before changes were made to it. Figure 9 shows how this is implemented.

The **save** command, also not available in the first OSMagic, allows a user to save the current version of a design and then continue working on it. In Magic this is implemented by writing the current version of the design to its corresponding flat file on disk and then allowing the user to continue work by not clearing the cell definition for that design from

the edit window. With this workspace model and the proper transaction model the **save** command can be implemented in OSMagic by first checking the current version of the design object into the parent workspace and then checking it back out again, thereby saving the current version in the global workspace and creating a new version of it in the user workspace for the designer to continue work. Proper transactions are needed to ensure the check-in and check-out operations are in different transactions since ObjectStore requires a check-in operation to be committed to the database before a check-out operation can be valid. Also the **writeall** command, which in Magic is basically the same as the **save** command except the edit cell is cleared afterwards, is implemented with a check-in to the global workspace of all cells modified since the last write or save operation. The cell is not immediately checked out again since the use of the **writeall** command indicates the user is done editing the current cell and wishes to clear the edit window.

3.4 Transaction Model - Stage 3

The fact that the first OSMagic had only one *main* transaction placed a size limitation on designs the tool could create and/or manipulate. To remove this limitation a transaction model had to be implemented which used multiple smaller transactions whose size is based on the smallest possible work units that maintain data consistency and integrity. As with the versioning model, our intent was to implement an initial transaction model which would allow for complete functionality of OSMagic as a single-user version of Magic and then build on that model as needed to address the more complex multi-user concerns OSMagic introduces.

3.4.1 Retaining Persistent Addresses. One of the first considerations that must be resolved when implementing a transaction model is how addresses to persistent data are to be resolved. The main question to answer, especially when converting an existing application to get database support, is how pointers to persistent memory will retain their validity across database transactions. ObjectStore defines five categories of pointers that

can be present in an ObjectStore application (15). Pointers from

- transient memory to persistent memory,
- persistent memory to transient memory,
- transient memory to transient memory,
- persistent memory to persistent memory in another database, and
- persistent memory to persistent memory in same database.

Only the last category of pointers will retain their validity at the end of a transaction without special handling. All others are no longer valid at the end of the transaction. Therefore, it is important to take some sort of action to ensure pointers retain their validity across transactions.

With ObjectStore there are several ways to ensure pointers are valid when referenced. One is to use ObjectStore *references* as substitutes for pointers. Once established, ObjectStore will automatically resolve them to the proper address when they are referenced within a transaction. This option is good for new applications since the references would be defined in the data structures. It is also good for existing applications with relatively few cross transaction pointers since it wouldn't require too much effort to redefine the pointers as references. Another option is to use ObjectStore *local references* which are similar but must be resolved explicitly by the application. Of course the brute force option is to just reestablish all pointers in each transaction. All of these options would require major changes to an existing application which is being interfaced with ObjectStore. ObjectStore does provide an alternative. By using the `objectstore::retain_persistent_addresses()` function, the normal releasing of persistent space at the end of each transaction is not done. Instead, all persistent addresses are globally retained and thus the pointers assigned to those ad dresses remain valid. With this option the addresses are retained for the life of the process or until the `objectstore::release_persistent_addresses()` function is invoked.

While all the options have a performance cost in terms of database access times, the last option also introduces the possibility of running out of persistent space in an application which has many references to persistent data. This possibility is more prevalent in the retain-persistent-addresses case since none of the persistent addresses are released. In the options using references, only those pointers declared as references are retained and

all others are released at the end of each transaction. However, due to the major changes required in implementing the other options, the retain persistent addresses option is the most appropriate when converting an existing application like Magic to use ObjectStore. Magic makes use of many pointers to persistent memory in its various complex algorithms. To reestablish these pointers in each transaction or to redefine them all as one of the reference classes of ObjectStore would require an intense programming effort. For this reason the `objectstore::retain_persistent_addresses()` is used in OSMagic. For the size of designs created and manipulated by the Magic users here at AFIT, this approach is not a problem. It is conceivable, however, that designs could be developed which would use up all persistent space. Any conversion effort of this type must consider this possibility and decide, based on the application being converted, whether the retain-persistent-addresses option can be used or if an approach using references is required.

3.4.2 The Initial Effort. The first OSMagic was implemented using two database transactions, one for initialization and one for everything else. After the tool was initialized, the *main* transaction was started. All work on a design was done in that transaction with nothing being committed to the database until the tool was exited and the transaction ended. All commands used during the session were done within this one transaction. For our first effort, we wanted to reduce the transaction size such that each Magic command was implemented within one *command* transaction. This would assure that all data affected by the command would be locked by the transaction and that in case of failure the database would return to the state it was in just prior to the command being issued.

In Magic there is a function named `TxDispatch` whose purpose is to recognize when a command is issued, determine what the command is and its arguments, dispatch the command for execution, and update the appropriate windows. To implement our first transaction size reduction, we removed the previously declared *main* transaction and in its place implemented several smaller transactions in the TxDispatch function. The main function of concern was `WindSendCommand`, which is where the command interpreter invokes the appropriate driver function for implementing the current command. By wrapping this function in a transaction, all actions taken to implement the command are within

the same transaction. This insures that all data being manipulated by the command is locked and that none of the changes to that data are committed to the database unless all actions complete successfully. This implementation will also allow us to nest transactions in lower-level functions since none of the transactions are committed until the outer most transaction is committed. Such nesting may be necessary when addressing recovery issues. Several other functions which affect persistent data are also called in the TxDispatch function. These include the `WindUpdate` function which implements all required actions for updating the appropriate windows to reflect the results of the last command, and the `DRCContinous` function which implements the design rule checking algorithm. These functions had to be wrapped in transactions as well.

Testing of this first effort found that it is not the implementation that will be required to make OSMagic a fully functional version of Magic. There are still several commands and situations which this implementation does not support. The **save** and **flush** commands still don't work and the writing of a new design that was created in the UNNAMED work cell does not work. The situation with the problem areas mentioned here are all related to the same problem. Each of them requires an adjustment to the global workspace version of the object being worked on, followed by a subsequent check-out operation of that object from the global workspace to the user workspace. In the case of the **save** command, the version of the object being modified has to be checked-in to the global workspace in order to freeze (save) that version. Then the object must be checked-out again in order that the user can continue work. The **flush** command requires destroying a version in the global workspace followed by a subsequent check-out operation in order to return to the previous version of the object. The writing of new designs fall into this same problem area because the UNNAMED cell itself is not versioned in the global workspace but instead is copied into a new cell definition which is. Currently the function which creates new cell definitions checks the initial version into the global workspace. It is then checked out again so the definition in the UNNAMED cell can be copied into it.

The current release of ObjectStore requires that an operation which adjusts an object's version hierarchy in the global workspace be committed to the database before the subsequent check-out of that object which adds a new version to that object's hierarchy.

This means that such operations must be in different transactions. We attempted to conform to this requirement by wrapping all check-in and check-out operations in separate transactions as well as the destroy operation used for the **flush** command. Since these operations occur in the code that implements each command, the new transactions were nested within the outer transaction wrapping the `WindSendCommand` function. Subsequent testing proved that the same problems persisted. Further consultation with the support section of Object Design found that actions which affect an object's version hierarchy must be truly committed to the database before the subsequent creation of a new version of that object. This means that not only must these actions occur in separate transactions, but that the transactions must be at the top level as well.

These problems show that our first attempt does not work as implemented. The transaction model will have to be changed such that appropriate transactions are defined in each command's driver function and any lower-level functions as the situation dictates for particular commands. This will ensure those actions which affect an object's version hierarchy are in top-level transactions. To properly implement this will require a detailed analysis of how each command is implemented so that proper transactions can be defined. Such an implementation will be a major undertaking which, because of time constraints, cannot be started by this thesis effort. While it was anticipated that a more complex transaction model would be required to address the areas of failure recovery, data consistency, transaction efficiency, and the roll back capabilities of Magic, it was felt the immediate concerns of a single-user implementation could be satisfied first. However, the level of effort foreseen to implement transactions in each command driver may be extensive enough to warrant the design and implementation of the full multi-user transaction model instead of the time it would now require to implement it in stages.

3.5 A Multi-User Versioning Model - Stage 4

If the only objective of this research was to implement a *single-user* version of Magic using ObjectStore, then the workspace model defined earlier would be sufficient. However, since the objective is to eventually develop a *multi-user* version of the tool which takes advantage of the capabilities a database provides, a more complex workspace model which

incorporates the design philosophy of the tool's users is required. We not only want to demonstrate that the Magic tool can be interfaced with an object-oriented database but that the added capabilities gained by that interface will outweigh any performance degradations. To achieve this objective a workspace model which fully addresses these added capabilities, and the design environment in which the tool is to be used, is required.

3.5.1 The Tool's Environment. The first step in determining the workspace model needed was to meet with the local users of the Magic tool, and find out how they used the tool. How these users controlled access to their design files, what procedures they followed when creating designs, how they handled multiple users working on the same design or parts of the same design, and similar questions were posed. The answers to these questions were crucial to obtain an understanding of the design environment.

Magic uses flat files which define the VLSI design objects Magic manipulates. These files makeup the database Magic accesses. Here at AFIT, the databases are implemented as unix directories with access controlled by unix directory and file permissions. These databases are referred to as design object libraries which fall into the following three categories:

1. A global library of designs and design objects which are considered in a stable or released state. Access to this library is controlled by the database manager which is the AFIT VLSI instructor.
2. A project library of designs which are assigned to particular teams of student designers. Access to this library is controlled by the team leader.
3. Multiple user libraries. Each student can define as many unix directories as they wish to hold copies of the design objects on which they work. They control access to their own libraries.

Control is implemented by setting privileges such that all have read permission to the global and project libraries but only a few have write permission. The user library permissions are under complete control of the user. Just who works on what and what modified objects are allowed back into the higher libraries is manually controlled via work assignments and cooperation between the student designers. When multiple users are to pursue work on the same design, the instructor controls which work is accepted for the library, if any, or whether the works are to be merged before being placed in the library.

The bottom line is that control of objects and databases is controlled through manual means.

As discussed earlier, Magic designs are defined as cells which are made up of multiple subcells. The prevailing design philosophy used here is that no two designers will work on the same subcell at the same time. A design is split into mutually exclusive pieces usually defined by major subcells. Designs are built in a bottom-up manner by aggregating previously designed subcells together. Of course maintenance and improvements will continue on subcells or changes may be required to larger designs. This often leads to cases where a designer or team of designers will be working on a particular subcell while another designer or team is working on a subcell at a higher level of the hierarchy which incorporates the other subcell. This requires permissions to be set which enable each designer or team write access to the pieces they are changing and read access to shared pieces. When either part is completed, the other designer or team must be informed so they can see any changes made and determine if there are any impacts on their work.

3.5.2 A Proposed Model. For OSMagic to support this design environment, the versioning model will require at least three classes of workspaces. A *global* workspace is needed to emulate the library of cell definitions which are considered stable. The model should allow anyone to check design objects out of this global workspace, but restrict check-in operations to those designated as the administrators of that library. To implement the library of team projects, a *project* workspace is required with permissions set which allow any designer assigned to that project to check-out design objects but restrict checking in of objects to the project administrator(s). The model must allow for designers to be assigned to multiple projects. Of course, each designer needs a *user* workspace in which to perform their individual work on subcells and designs. This class of workspace must be the most restrictive in that only the user who owns the workspace can see the objects in it. It will be in the *user* workspace where the original single-user Magic is implemented.

Implementation of this model requires a means of establishing the workspace hierarchy for each user of OSMagic, and commands which allow the user to adjust the current workspace in order to establish the proper parent child relationships for check-in and check-

out operations. When started, the application will have to determine who the user is and what privileges they have when establishing the initial workspace relationships. This could be accomplished by defining hash tables which are maintained in the database to track the user-ids of all valid users and the privileges they have for the different workspaces. New commands for updating these tables would be required which are only accessible to the database administrator.

With such tables defined, the application would then use login ids to identify the user of the tool and establish the initial workspace relationships. Whenever an attempt is made to check an object into the project or global workspaces, the user's privileges would be checked to see if the operation should be allowed. Commands will also be required which allow users to adjust the workspace relationships when needed to allow for check-out operations from either the global or project workspaces. These are required since there may be cases when a user needs to check a design out of the project workspace to work on and then check out a subcell from the global workspace which will be added to the design. The added commands will be needed to establish the proper parent-child relationships for the check-in and check-out operations. This can be implemented by using ObjectStore commands to establish pointers to the desired workspaces and then setting the *current* workspace.

With the implementation of this proposed model and the transaction model needed to fully support it, OSMagic will support the cooperative work environment of the AFIT user community. The automatic concurrency control and failure recovery mechanisms of ObjectStore will allow multiple users access to the same data objects at the same time. Concurrent development of alternative design paths for the same design object will be supported through the use of ObjectStore's versioning facilities. With these models in place, OSMagic will fully support the multiple component libraries and workspaces used by the AFIT user community and incorporate their design philosophy in the use and control of these areas.

IV. Results Analysis

4.1 Overview

The primary objective of this thesis was to show that an existing computer-aided design tool can be supported by an object-oriented database management system without loss of functionality. We also wanted to show that any degradation in performance is compensated by the introduction of additional capabilities. Through the use of ObjectStore's version and transaction management facilities, the foundation for achieving this goal has been laid. In this chapter, we describe the achievements made in converting OSMagic to a point where it exhibits most of the original Magic's functionality and the lessons learned from the difficulties encountered in that effort. We also present a comparison of the performance of OSMagic in its current state and environment with that of the original Magic in the same environment.

4.2 Functionality of OSMagic

For complete and concrete conclusions to be made about OSMagic in comparison to Magic, it must be shown that OSMagic can perform the same, or compatible, functions as Magic. Until such a state is achieved, only intermediate results about specific areas of functionality can be analyzed and discussed. While this effort has not yet taken OSMagic to a state where it can perform ali functions of the original Magic, it has implemented the all important ground work necessary in order to achieve that state. Basic implementations of ObjectStore's version and transaction management facilities have been made in a manner which will not only lead to an OSMagic with all the functionality of Magic, but one which will also have added multi-user capabilities.

The current OSMagic will perform all functions of Magic with the exception of those which require multiple check-in/check-out operations to and from the global workspace. Because ObjectStore requires such operations to be in separate **top-level** transactions, these functions won't be available until a more complex transaction model is implemented. OSMagic now maintains and tracks versions of all VLSI designs stored in the database and manipulated by the tool. Once the proper transaction model is implemented, these versions

will allow designers the ability to roll back to a previous version of their designs. With its new versioning capabilities, OSMagic now supports the *long* transactions inherent in design environments, and provides the foundation needed to allow for alternative development paths of the same design objects by the same or different designers. Since it is common in a design environment to pursue several design alternatives for a particular design object, this is a very important feature for multi-user tools. OSMagic is now implemented with multiple transactions instead of the two transaction implementation of the first OSMagic. While the current transaction model falls short of that required for a fully functional implementation, we believe it removes the size limitation of designs that can be manipulated by the tool, although we have not run tests to confirm this. Such tests should be conducted after the more complex transaction model is implemented.

The problems of improper allocation of persistent data objects and the persistent definition of internal data structures which should remain transient have been corrected in the current version of OSMagic. In the process of identifying and correcting these problems, several valuable lessons have been learned. First is the fact that when converting existing CAD tools to database implementations, it is highly probable that there will be data structures of the same data type that must be allocated in both transient and persistent memory. In this conversion of Magic to OSMagic the cell definition and hash table data structures are examples of such data types. In this case, once the appropriate allocation and reference mechanisms were implemented for these data structures, there were no adverse impacts on the tool's algorithms. By allocating both persistent and transient hash tables and using flags to indicate which is the appropriate table for the current invocation, all hashing functions work as before. With one exception, the same is true for those functions which manipulate the cell definition structures in OSMagic. As explained in Chapter 3, we did have to modify the definition of the *tile* data structure to implement the multi-purpose pointer fields as C `unions`. We also had to modify those functions which reference these pointers to use the appropriate field of these unions. Because this situation is most likely common to all computer-aided design tools, any effort to convert such tools to use a database will have to address similar problems. Any such conversion will require a thorough analysis of all data types to determine if any of those types require allocation in

both transient and persistent memory space and what, if any, impact there will be on the tool's algorithms because of that requirement.

We also learned that many of the programming techniques used in *transient memory based* CAD tools either do not work or require considerable analysis and modification to implement in a *persistent memory based* implementation of that tool. Those techniques which take advantage of language-specific *side effects* to implement programming shortcuts, efficiency gains in algorithms, or for other reasons may work in transient memory space but can give completely different results when used in persistent space. Such programming techniques are not good practice by any standard and should be eliminated. The use of the C programming language *casting* facility to implement the utility pointers in Magic's data structures is a example of this type of technique which we had to correct. As detailed in AppendixD, the proper data structure for this situation was the C `union`. There are also many valid programming techniques whose implementation for a transient-based application require modification if they are to work properly in a persistent-based conversion of that application. A good example of this is the abundant use of flag and bit-masks fields. In Magic such fields are used in every algorithm. The problem we encountered was the fact that these fields, which are transient in Magic, are now persistent in OSMagic. In Magic they are established and initialized when cell definitions are created and go away at the end of the process since they are not saved in the flat file that cell definition represents. In OSMagic however, the non-internal cell definitions are now persistent, making these flag and bit-mask fields persistent as well. This requires modifications to insure these fields, which are used to control most algorithms in the tool, are placed in a proper state when a session ends. In our case, because of the abundant use of such fields, this required an intense analysis effort. The impact of such programming techniques can be considerable in terms of the analysis and modification effort required in addressing them. This is especially true for those techniques which make use of language-specific or machine-dependent side affects to implement a software algorithm. The elimination of such techniques from existing CAD tools, which are intensive and complicated applications, can be costly and time consuming. Careful analysis is required to choose the approach most appropriate to the situation at hand. Many programming techniques and practices which were quite valid for

the environments such tools were written in may not be conducive to the environments of the database implementations of those tools. Again, this situation is probably a common trait of most existing CAD tools and therefore is a concern of any conversion of such tools to a database implementation.

With the data allocation and type definition problems corrected, the first versioning model implementation was completed. The model includes global and user workspaces which, when implemented with a proper transaction model, will insure OSMagic is a fully functional implementation of Magic. Testing of this model made clear which commands worked and which did not. Analysis as to why certain commands still don't work not only pointed to the need for a more complex transaction model, but also pointed out another valuable lesson. We designed the versioning model by making use of ObjectStore's versioning facilities to implement the commands of Magic. For the most part this is the proper approach, especially when trying to get a working prototype. However, our design of the model should have recognized the fact that there are cases where the implementation of the commands should be changed to fit the model rather then the model designed to fit all commands. An example of this is the implementation of the **save** command where instead of checking a version into the global workspace and checking a new version out for continued work, the versioning should all be done within the user workspace. When implemented in this manner, a check-in to the global workspace is not required until the user feels the design is in a more stable state. In any conversion effort of this type, such cases must be identified and included in the design of the versioning and transaction models.

With the basic versioning model in place, we implemented the first phase of our transaction model by wrapping all actions required for each of the tool's commands in a transaction. Then nested transactions were defined for those commands requiring check-in and check-out operations for versioning purposes. Recovery concerns were planned for the second phase of the implementation. Testing discovered that the use of nested transactions still didn't allow those commands requiring the versioning actions to work. Object Design's support branch confirmed this finding and indicated the current requirement for such actions to occur in top-level transactions was a limitation in the current release of ObjectStore which they plan to correct in a future release. Another valuable lesson was

learned here that again applies to any similar conversion effort. It is quite possible that the transaction and versioning facilities of the OODBMS being used will not allow the most optimal implementation of the models developed. In our case this limitation means that a much more complex transaction model is required. Such a model requires changes to many lower-level functions which previously had not required modification. In OSMagic this problem gets magnified since these lower-level functions are in C modules which must be converted to C++ if we are to maintain our secondary objective of using separate interfaces to ObjectStore. If such limitations can be identified early, time and costs of a conversion effort can be saved by developing models accordingly.

4.3 Current OSMagic vs Magic - Performance Comparisons

Probably the single most important consideration that will judge the worthiness of an effort to convert existing computer-aided design tools to implementations on object-oriented databases is that of performance. Complex engineering design tools demand high performance requirements, which is why such tools have not had the support of database management systems in the past. Because of the incomplete state of the current OSMagic implementation, extensive testing has not been done. Testing has been accomplished, however, for those commands which were used in the testing of the first OSMagic effort, with the exception of the closure test using the **drc catchup** command since that command requires the more complex transaction model implementation. An effort is made to interpret the results of these tests in light of the versioning and transaction models currently in place.

4.3.1 Description of Tests Performed. The design objects used for these test are the **drfmchip** and **tut4a** chip which were described in detail in the work by Jacobs (8). The commands tested are as follows:

load cellname This command tests the look up and retrieval of an object from the database. In Magic the command searches the appropriate unix directory for the file representing the given cellname and uses the instructions in that file to create the appropriate cell definition and display it the selected window. In OSMagic, the com-

mand searches the database for the cell definition with that name, checks it out of the global workspace into the user workspace if it isn't already checked out, and displays it in the selected window. For these tests, the cells have been stored such that the load operation will display them in their unexpanded form.

expand This command tests the traversing of the pointer hierarchy of the edit cell. The command will retrieve and display all subcells which are in a boxed area of the edit cell. By boxing in the entire edit cell, all subcells of that cell are retrieved and displayed in the selected window. The cells are retrieved in the same manner as the load command.

getcell This command tests the insertion of a data object into the hierarchy of the current edit cell. It is used to test such an insertion at three different subcell hierarchy levels. It takes as an argument the name of the cell which is to be inserted, finds that cell in the database, creates an instance of it, and inserts that instance in the appropriate level of the edit cell's hierarchy of subcells.

The insertion tests conducted with the **getcell** command were done the same as in the work by Jacobs (8). With the *drfmchip* cell, the **getcell** command was tested four levels deep in the *big_nandmux* subcell and eight levels deep in the *mcell10* subcell. It was tested at the top level only with the *tut4a* cell. As in Jacobs (8), the measurements for writing modified cells back to disk were added to the insertion tests of the original Magic to provide a more accurate comparison with the OSMagic tests. The *Initialization* measurements reflect those database operations required for initializing memory.

The data for these test was gathered using the performance hooks implemented by Jacobs (8) in the first OSMagic prototype. These performance hooks call the **Command-Stats** routine which makes calls to the **UNIX** *getusage* and *gettimeofday* functions in order to calculate the response time, page faulting, and I/O statistics desired.

Tests were conducted in OSMagic with all data objects located in the global workspace, thereby requiring a *check-out* operation to get the data into the user workspace, and with all data already in the user workspace so that no *check-out* was required. These tests and the tests of the original Magic were conducted on a SUN Workstation with 32MB of RAM.

Criteria Tested *Command Used*	Resource Measured	Data Management System		Percent Change
		Flat File	ObjectStore	
Initialization	CPU time (seconds)	5.822	6.517	+12
	Elapsed time (seconds)	11.178	14.558	+30
	Page Faults with I/O	41*	62	+51
	Page Faults without I/O	131	419	+219
	Disk Blocks In	14*	13	-7
	Disk Blocks Out	1	1	0
Look up/retrieve *load drfmchip*	CPU time (seconds)	0.078	1.162	+1390
	Elapsed time (seconds)	0.144	4.253	+2853
	Page Faults with I/O	1*	14	+1300
	Page Faults without I/O	8	238	+2875
	Disk Blocks In	0	0	0
	Disk Blocks Out	0	0	0
Hierarchy Traversal *expand*	CPU time (seconds)	7.083	134.98	+1806
	Elapsed time (seconds)	17.480	270.63	+1448
	Page Faults with I/O	146	898	+512
	Page Faults without I/O	103	3664	+3457
	Disk Blocks In	146	1	-99
	Disk Blocks Out	0	0	0
Insert (instance) *getcell test* 4 levels of nesting	CPU time (seconds)	0.073	1.203	+1548
	Elapsed time (seconds)	1.636	2.267	+38
	Page Faults with I/O	0	7	$+\infty$
	Page Faults without I/O	16*	203	+1169
	Disk Blocks In	0	0	0
	Disk Blocks Out	1	0	$-\infty$
Insert (instance) *getcell test* 8 levels of nesting	CPU time (seconds)	0.068	1.617	+2278
	Elapsed time (seconds)	1.446	3.595	+149
	Page Faults with I/O	2	18	+800
	Page Faults without I/O	5	245	+4800
	Disk Blocks In	1	0	$-\infty$
	Disk Blocks Out	1	0	$-\infty$

* - average may not accurately reflect raw results

Table 1. Benchmark performance results for **drfmchip** in Global Workspace

Tests were also conducted in OSMagic, with all data located in the global workspace, on a SUN Workstation with 48MB of RAM. Tables 1 and 2 presents a comparison between the compiled test results for Magic vs the compiled results for OSMagic with all data in the global workspace on the 32MB RAM workstation. Tables 3 and 4 show the same

Criteria Tested *Command Used*	Resource Measured	Data Management System		Percent Change
		Flat File	ObjectStore	
Initialization	CPU time (seconds)	5.808	6.725	+16
	Elapsed time (seconds)	10.342	21.935	+112
	Page Faults with I/O	20*	169	+745
	Page Faults without I/O	153*	228	+49
	Disk Blocks In	19	21	+10
	Disk Blocks Out	1	1	0
Look up and retrieve *load tut4a*	CPU time (seconds)	0.017	0.617	+3529
	Elapsed time (seconds)	0.050	1.884	+3668
	Page Faults with I/O	1	9	+800
	Page Faults without I/O	7	81	+1057
	Disk Blocks In	0	0	0
	Disk Blocks Out	0	0	0
Hierarchy Traversal *expand*	CPU time (seconds)	0.032	0.968	+2925
	Elapsed time (seconds)	0.119	2.531	+2027
	Page Faults with I/O	1	5	+400
	Page Faults without I/O	2	123	+6050
	Disk Blocks In	1	0	-∞
	Disk Blocks Out	0	0	0
Insert (instance) *getcell test* Top level	CPU time (seconds)	0.055	0.932	+1594
	Elapsed time (seconds)	1.907	2.097	+10
	Page Faults with I/O	1	4	+300
	Page Faults without I/O	5	125	+2400
	Disk Blocks In	1	0	-∞
	Disk Blocks Out	1	0	-∞

* - average may not accurately reflect raw results

Table 2. Benchmark performance results for **tut4a** in Global Workspace

comparison between Magic and OSMagic with all data in the user workspace on the 32MB workstation.

These tables show the averages of the results for each command. Those values flagged with an * indicate averages which may not reflect the raw data as well as desired due to wide fluctuation in results received. The results of the raw performance tests are contained in Appendix E.

The results of a comparison of OSMagic's performance when run on a SUN Workstation with 32MB internal RAM against its performance on a SUN Workstation with 48MB of RAM is shown in Table 5. In these tests all data was in the global workspace.

Criteria Tested *Command Used*	Resource Measured	Data Management System		Percent Change
		Flat File	ObjectStore	
Initialization	CPU time (seconds)	5.822	6.730	+16
	Elapsed time (seconds)	11.178	14.163	+27
	Page Faults with I/O	41*	15*	-63
	Page Faults without I/O	131	451	+244
	Disk Blocks In	14*	3*	-79
	Disk Blocks Out	1	1	0
Look up/retrieve *load drfmchip*	CPU time (seconds)	0.078	0.862	+1005
	Elapsed time (seconds)	0.144	2.305	+1501
	Page Faults with I/O	1*	1*	0
	Page Faults without I/O	8	220	+2650
	Disk Blocks In	0	0	0
	Disk Blocks Out	0	0	0
Hierarchy Traversal *expand*	CPU time (seconds)	7.083	22.79	+222
	Elapsed time (seconds)	17.480	98.49	+463
	Page Faults with I/O	146	73*	-50
	Page Faults without I/O	103	3046	+2857
	Disk Blocks In	146	0	-∞
	Disk Blocks Out	0	0	0
Insert (instance) *getcell test* 4 levels of nesting	CPU time (seconds)	0.073	1.247	+1608
	Elapsed time (seconds)	1.636	2.141	+31
	Page Faults with I/O	0	4	+∞
	Page Faults without I/O	16*	235	+1369
	Disk Blocks In	0	0	0
	Disk Blocks Out	1	0	-∞
Insert (instance) *getcell test* 8 levels of nesting	CPU time (seconds)	0.068	1.187	+1646
	Elapsed time (seconds)	1.446	1.550	+7
	Page Faults with I/O	2	0	-∞
	Page Faults without I/O	5	210	+4100
	Disk Blocks In	1	0	-∞
	Disk Blocks Out	1	0	-∞

* - average may not accurately reflect raw results

Table 3. Benchmark performance results for **drfmchip** in User Workspace

4.3.2 Interpretation. When comparing tests results, care must be taken to understand, to the fullest extent possible, all factors which have a bearing on those results and any additional considerations which must be weighed in the interpretation of those results. While we cannot completely explain all influences reflected in our test results, we can point out some areas we know to have an impact. In our case, we must remember that additional

Criteria Tested *Command Used*	Resource Measured	Data Management System		Percent Change
		Flat File	ObjectStore	
Initialization	CPU time (seconds)	5.808	6.730	+16
	Elapsed time (seconds)	10.342	14.163	+37
	Page Faults with I/O	20*	15*	-25
	Page Faults without I/O	153*	451	+195
	Disk Blocks In	19	3*	-84
	Disk Blocks Out	1	1	0
Look up and retrieve *load tut4a*	CPU time (seconds)	0.017	0.655	+3753
	Elapsed time (seconds)	0.050	1.394	+2688
	Page Faults with I/O	1	1	0
	Page Faults without I/O	7	83	+1086
	Disk Blocks In	0	0	0
	Disk Blocks Out	0	0	0
Hierarchy Traversal *expand*	CPU time (seconds)	0.032	0.700	+2088
	Elapsed time (seconds)	0.119	1.053	+785
	Page Faults with I/O	1	0	-∞
	Page Faults without I/O	2	89	+4350
	Disk Blocks In	1	0	-∞
	Disk Blocks Out	0	0	0
Insert (instance) *getcell test* Top level	CPU time (seconds)	0.055	0.905	+1545
	Elapsed time (seconds)	1.907	1.246	-35
	Page Faults with I/O	1	0	-∞
	Page Faults without I/O	5	107	+2040
	Disk Blocks In	1	0	-∞
	Disk Blocks Out	1	0	-∞

* - average may not accurately reflect raw results

Table 4. Benchmark performance results for **tut4a** in User Workspace

capabilities have been introduced in OSMagic at a cost of additional overhead. For that reason, the trade-offs between those capabilities and their overhead costs must be considered when comparing the performance results of OSMagic to those of Magic. In the current environment a client-server relationship exists between the ObjectStore *client* processes on the SUN workstations and the ObjectStore *server* process on the NFS network server. All ObjectStore client processes have default client caches of 4MB. This means there is 4MB less of available memory to OSMagic, thus increasing the amount of page faults required. We must also consider the amount of overhead associated with the use of ObjectStore in order to implement the added capabilities it provides.

Criteria Tested *Command Used*	Resource Measured	Memory Size (RAM) 32MB	48MB	Percent Change
Initialization	CPU time (seconds)	6.517	6.705	+3
	Elapsed time (seconds)	14.559	19.858	+36
	Page Faults with I/O	62	142	+129
	Page Faults without I/O	419	316	-24
	Disk Blocks In	13	20	+54
	Disk Blocks Out	1	1	0
Look up/retrieve *load tut4a*	CPU time (seconds)	0.620	0.638	+3
	Elapsed time (seconds)	1.531	2.150	+40
	Page Faults with I/O	4	5	+25
	Page Faults without I/O	94	92	-2
	Disk Blocks In	0	0	0
	Disk Blocks Out	0	0	0
Look up/retrieve *load drfmchip*	CPU time (seconds)	1.162	1.143	-2
	Elapsed time (seconds)	4.252	3.647	-14
	Page Faults with I/O	14	0	-∞
	Page Faults without I/O	238	258	+8
	Disk Blocks In	0	0	0
	Disk Blocks Out	0	0	0
Hierarchy Traversal *expand drfmchip*	CPU time (seconds)	134.752	129.918	-4
	Elapsed time (seconds)	270.632	316.634	+17
	Page Faults with I/O	898	14*	-98
	Page Faults without I/O	3664	4787	+31
	Disk Blocks In	1	0	-∞
	Disk Blocks Out	0	0	0
Insert (instance) *getcell test* 8 levels nesting	CPU time (seconds)	1.617	1.598	-1
	Elapsed time (seconds)	3.597	4.636	+29
	Page Faults with I/O	18	1	-94
	Page Faults without I/O	245	259	+6
	Disk Blocks In	0	0	0
	Disk Blocks Out	0	0	0

* - average may not accurately reflect raw results

Table 5. ObjectStore performance comparison with two different RAM sizes

The tests of OSMagic with all data initially in the global workspace, as reflected in Tables 1 and 2, show performance degradations in almost all areas in comparison to Magic. The most disturbing and noticeable difference is in area of hierarchy traversal where elapsed time increased from approximately 17 seconds to 270 seconds and the amount of page faulting without I/O went from 103 to 3664. This is certainly unacceptable by any

standard. The results are much better for the tests of OSMagic with all data initially in the user workspace, as seen in Tables 3 and 4. In this case, OSMagic performs as well as or better than Magic in over half of the areas tested. With few exceptions, OSMagic, with the data in the user workspace, performed better than it did with the data in the global workspace, especially in the area of hierarchy traversal. This improvement can be directly contributed to the absence of movement between workspaces. When no check-out operations are required, the overhead of the versioning tasks associated with such operations is eliminated. However, the performance degradation in the area of hierarchy traversal is still noticeable and possibly unacceptable. While much better than the case with the data in the global workspace, we still have an increase from an average of 17 seconds of elapsed time in Magic to approximately 98 seconds in OSMagic, and an increase from 103 page faults without I/O to 3046 page faults. The increased time is clearly due to the amount of page faulting taking place. The increased page faulting can be partly explained by the fact that OSMagic has less available memory due to ObjectStore overhead and by the caching used in ObjectStore's memory mapping architecture. Beyond that, a means of throughly tracking memory usage by both OSMagic and Magic is needed to completely explain the results.

While our results show some high percentages of change, most response time degradations are not that drastic when viewed relative to the the users perception of them. Nonetheless, the numbers do reflect performance degradations and it must be decided whether those degradations are warranted by the increased capabilities the database brings to the tool. If the response times in OSMagic are acceptable to the users of the tool in light of the added versioning and multi-user user capabilities, then one can say the implementation is good. If, on the other hand, the performance is not acceptable, then the conversion is bad. Another important consideration is the fact that OSMagic is not yet in an optimal state. So if the results we are seeing now are acceptable in most areas, then it is highly probable they will be better in a fully implemented OSMagic and that those areas considered unacceptable now will become acceptable in the fully functional and optimized implementation.

We should also point out that the tests of OSMagic with all data initially in the user workspace are probably the most relevant in a comparison with Magic. The normal use of a fully implemented multi-user OSMagic will have a designer checking a design object out of either the global or project libraries into their user workspace where they will work on it. The design will remain in their workspace until they have completed all work on it and are ready check it back into the appropriate library workspace. It is in the user workspace where the functionality of the original Magic will be implemented. Therefore the closest possible comparison of OSMagic and Magic is the case when all data is located in the user workspace.

In a design environment, it is normal for design work to take a long period of time. This means the case of checking objects out of the global or project workspaces will be a relatively infrequent event, only occurring when a design is initially obtained and when work on it is complete. Therefore the performance degradation due to the overhead incurred in versioning operations between workspaces will not be a significant problem. Also, it is possible that the use of ObjectStore subconfigurations could significantly reduce this performance degradation since they can greatly reduce number of check-out operations required depending on the implementation. However, the use of subconfigurations could have an adverse impact on the implementation of data object libraries. The trade-offs would have to be weighed carefully.

We wanted to see what impact, if any, a larger RAM would have on the performance of OSMagic. The tests done with all data in the global workspace were repeated with an ObjectStore client process set up on a 48MB RAM workstation. As the results in Table 5 show the only real conclusion that can be drawn is that the decreased amount of page faulting with I/O was accompanied by an increased amount of page faulting without I/O.

4.4 Interface and Environment Issues

Secondary goals of this thesis were to create an environment which would enhance the programming and debugging efforts required in the conversion and to explore the differences between implementations of OSMagic using the three ObjectStore interfaces. Because of the problems encountered, unresolved issues, and most of all time constraints,

neither of these goals were completely realized However, as with the main efforts, the necessary foundation for implementing and exploring these goals has been laid.

The code for all ObjectStore commands has been implemented in all three ObjectStore interfaces. Through the use of preprocessor variables and the appropriate compiler options, OSMagic can now be compiled and linked to use the C++ and DML interfaces to ObjectStore. With the implementation of appropriate makefile changes and work-arounds for the ObjectStore commands not yet available in the C library interface, OSMagic will be able to use ObjectStore's C library interface as well. One enlightening lesson learned in this effort was the fact that ObjectStore's DML interface is implemented using an extra level of indirection hidden to the programmer. This will have no impact on a normal conversion unless a mixture of the ObjectStore interfaces is used and an attempt is made to use pointers established with commands from one of the library interfaces to access data stored using DML persistent variables. The most common approach in a conversion, however, will be to choose an interface to ObjectStore and code everything in that interface, in which case this is not an issue. This is an issue, however, when multiple tools are involved which use different interfaces to ObjectStore and one of those interfaces is the DML. Then the indirection introduced by the DML interface is an issue which must be addressed.

With OSMagic implemented using the C++ interface to ObjectStore, we then worked at getting ObjectCenter to load OSMagic. The necessary environment initialization file was created and appropriate targets added to the OSMagic top-level makefile. However, we were unable to get OSMagic working in the ObjectCenter environment. While we could get the application initialized and the display of the initial windows, any attempt to load designs into the windows resulted in ObjectCenter finding null pointers. We were unable to determine the cause of these errors. Nor were we able to determine to our satisfaction whether ObjectCenter would handle the way in which the OSMagic source code was laid out and be able to find the source when needed for debugging purposes. The problems we encountered in our attempts to use ObjectCenter are again likely to be encountered in similar conversion efforts. It is quite common to want to use programming environment tools to perform such conversions. The issues involved with interfacing third-party tools with the database being used and the application involved must be addressed. Careful up

front analysis of which tools work best together will greatly reduce the number of problems encountered in this area.

4.5 Effort Required

The time and effort required to bring OSMagic to its current state was extensive, with the difficulty factors discussed previously contributing heavily to that effort. The learning curves required for the Magic application and the ObjectStore database were much greater than initially anticipated because of the lack of in-depth documentation on both applications and the complicated algorithms of the Magic application. It is important to note that this research effort cannot be directly compared to an effort which may be undertaken in a real-world situation. First, the effort required in implementing functionality or capabilities solely for research purposes, like the investigation of the three ObjectStore interfaces in our work, must be factored out. Secondly, this effort is being worked by one researcher at a time instead of a team of personnel which would be most probable for a conversion effort on a tool of this size and complexity. Also, there will not be the start up times of this environment as one researcher leaves and another takes over. Finally, it is highly probable that the learning curves for the programming language, development environment, and functionality of the tool being converted will be substantially less for the team in a real-world scenario than for the students in a research environment. All of these factors must be considered when attempting to use research efforts like ours to determine the difficulty of a real conversion effort.

V. Conclusions and Recommendations

5.1 Overview

This thesis studied the feasibility of using an object-oriented database management system to provide the functionality and performance needed to support a complex computer-aided design tool. Version and transaction management models were designed and implemented in the OSMagic prototype. These models now provide the foundation for making OSMagic a fully functional multi-user prototype of Magic which supports a cooperative design environment. In this chapter we present our conclusions based on the results presented in Chapter 4 and summarize the important lessons learned about conversions of this type. Finally, recommendations are presented for future research which will further the goals of this research.

5.2 Conclusions

The primary objectives of this thesis required the design and implementation of basic version and transaction management models in OSMagic. Such models are needed to make OSMagic a fully functional implementation of Magic which also has added multi-user capabilities. The secondary objectives required instrumentation of code to allow for development in the third-party programming environment, ObjectCenter, and for the implementation of OSMagic using all three interfaces to the ObjectStore OODBMS. The discovery of problems in the first OSMagic prototype required detailed analysis of the database code implemented in that prototype.

5.2.1 Functionality. OSMagic performs all functions of the original Magic with the exception of those requiring the more complex transaction model needed to support the necessary versioning actions in those functions. Basic versioning and transaction models are in place which provide the foundation required for OSMagic to be a fully functional, multi-user implementation of Magic. Problems encountered with the definition of data types and the allocation of persistent data objects have been corrected and all illegal addresses removed from the database. Code is in place to allow OSMagic to use the

different ObjectStore interfaces and to allow for future development using the ObjectCenter programming environment.

The basic versioning model is in place with the necessary workspaces and configurations it requires defined and implemented. Designs created and manipulated in the tool are now versioned and those versions tracked by the ObjectStore version hierarchy mechanisms. Stable versions of the designs are now maintained in the *global* workspace and a *user* workspace is used for development work on those designs. Most of the commands available in the original tool are now implemented and work with this versioning model. A basic transaction model was designed and the initial phases implemented. Limitations in the current ObjectStore implementation require a more complex transaction model to be designed and implemented for the prototype to have all the functionality of the original tool.

Before the implementation of the version and transaction management models could be completed, it was necessary to correct the problems of data allocation and data type definitions discovered in the first prototype. Until the basic groundwork of properly defined and allocated data structures is in place, the more complex management models couldn't be implemented. These corrections required an intense analysis and coding effort which ultimately used up the time needed to implement the more complex transaction model required for the prototype.

The OSMagic prototype can be implemented using either the C++ or DML interfaces to the ObjectStore OODBMS. Preprocessor variables are in place which, when defined through the appropriate compiler option, will ensure the corresponding interface code is compiled and linked in an executable image of the prototype which uses the desired interface to the database. There are some ObjectStore functions not yet available in its C Library interface. With the defining of the appropriate bindings to the implementation of these functions in one of the other interfaces, and the appropriate makefile adjustments, OSMagic will be able to use the C library interface to the database as well. All code for the interface is in place.

5.2.2 Performance Results. OSMagic performs as well or better than the original Magic in almost half of the tested areas when used in a scenario which most closely matches the normal operating scenario of the original Magic. As expected, the amount of input or output for the servicing of read and write requests was zero in all but the initialization cases and the number of major page faults (those requiring physical I/O) was significantly less. These results reflect the benefits of a database management system. The amount of minor page faulting (those not requiring physical I/O) increased significantly. This most probably reflects the amount of caching done by ObjectStore. There is also an across-the-board slowdown in response times. Except for the area of hierarchy traversal, the increased response times are not that dramatic. In most cases these slowdowns will not be perceived by the user and when considering the added capabilities in a database implementation, are acceptable tradeoffs. This same conclusion will hold for the area of hierarchy traversal as well when we consider the normal use of the command used in testing this functional area. In OSMagic the *expand* command provides the best test of the traversing of the hierarchy of pointers which define a design object. For our tests we used this command to **fully** expand the chip designs used as our tests cases. The full expansion of the smaller tutorial chip resulted in an approximate increase of one second response time. This is in line with the acceptable slow downs of the other tested commands. The full expansion of the much larger and more complex *drfmchip* design resulted in a much more perceptible increase in response time of approximately 81 seconds. However, the full expansion of a chip design with such complexity is a rare event. The normal situation is for expansions of smaller portions of the design. In this case the response time increases are much less noticeable.

When we consider the normal use of the tool's commands, the impact of any performance degradations as perceived by the user, and the benefits of the multi-user implementation of the tool with the added database management system capabilities, we can conclude that the performance results are very positive. In light of these results and the expected improvements in performance as a result of a fully implemented and optimized tool, we can conclude that an object-oriented database management system can provide the capabilities and performance needed to support typical computer-aided design tools.

5.2.3 Valuable Lessons. Throughout this effort we attempted to identify concepts which might apply to all conversion efforts of the type we undertook in this thesis. We feel these are important concepts which must be understood and handled in any such conversion effort. An understanding of these concepts as they apply to OSMagic and the impact they had on our conversion effort, will provide valuable insights which can be used to reduce the impact these areas have in similar conversion efforts. The following is a summary of what we feel are the valuable lessons learned during our conversion effort. Each was discussed in more detail in Chapter 4.

- There will most likely be data structures of the same data type that will require allocation in both persistent and transient memory space. This impacts the algorithms which manipulate those data structures by requiring code to identify which type of allocation applies in any particular case and ensuring the appropriate actions are taken. By identifying such data structures early, the actions needed to support them can be planned for early in the design phase of the conversion effort.
- Some programming techniques used in the implementation of the original tool being converted, like the abundant use of flags and bitmasks, may not be conducive to an implementation in persistent memory space. This is especially true for those techniques that take advantage of language-specific side effects like that of the C casting facility used in Magic. Adjustments will be required in the algorithms using such techniques and there will be an impact in terms of the time and effort required for the conversion. Again, early detection of these kinds of programming techniques will allow for proper planning in the design phase and reduce the impact they will have on the conversion effort.
- Version and transaction models needed for a conversion should not be designed such that they conform to every command or function of the tool. Rather, they should be designed in a manner which best supports the tool overall with the necessary modification of commands and functions to support that design. This approach will ensure a more efficient and effective model is implemented and eliminate the need for extraneous *special* work-around code which can be a real maintenance problem.
- To the extent possible with available documentation and support services, the limitations of the DBMS being used for the conversion should be identified early and accounted for in the design phase of the effort. Unanticipated limitations can and will impact the conversion in terms of the time and effort required. Such limitations may prevent the implementation of the most optimal conversion approach. By identifying them early, the most optimal approach with the limitations in mind can designed and the need for redesign or work-arounds can be avoided.
- It is important to determine early on the action required to ensure pointers retain their validity across database transactions. An analysis of the application being converted must be conducted to determine the likelihood of running out of persistent

address space. Based on this analysis the proper method for retaining persistent addresses can be implemented.

- Interface concerns for third-party tools should be identified and accounted for early in the conversion effort. Such tools may be used to aid in the conversion effort or may be part of the same design environment and work with the same database. It is important to ensure the interfaces each of the tools use to access the database are compatible. If special actions are required to ensure compatibility of all interfaces to the database, as is the case with the DML interface of ObjectStore, then the interfaces used in tools already implemented on the database may restrict the choice of interface used in the conversions of other tools to an implementation on that same database.
- Performance tests should be developed which will provide the best possible comparison with the operating scenario of the original tool. Analysis of test results must consider the difference in the operating scenarios, the added capabilities and functionality in the new tool, and the perception of the users of the tool. The addition of a database management system will without doubt add overhead which results in performance degradations in comparison to the original tool as will the models which implement the added functions in the new version of the tool. These things must be properly understood and considered when drawing conclusions about the results of a conversion effort.

5.3 Recommendations

OSMagic is not yet a fully functional and optimized implementation of Magic. Until it is brought to this completed state, a complete and detailed analysis of the benefits and drawbacks of such a conversion effort cannot be accomplished. The following are recommendations for areas of further research which we feel are needed to bring the conversion of Magic to OSMagic to a completed state:

- The design and implementation of a transaction model which will allow OSMagic to be a fully functional implementation of Magic. This model should be designed to support the versioning model of the multi-user version of the tool.
- The implementation of the more complex versioning model which will support the design environment of the local user community. This will include a workspace model which supports the design libraries of the user community and the merging of alternative designs developed by the same or different designers.
- The completion of the C library interface to ObjectStore and the through testing of all three interface implementations to ascertain which interface to the database is best suited for conversions efforts of this type.

5.4 Summary

The performance of the current prototype of OSMagic does not by itself justify the conversion of existing computer-aided design tools to implementations on object-oriented database management systems. But when viewed in light of the additional functionality and capabilities a fully implemented and optimized conversion is expected to have, there is reason for positive expectations that such a justification will eventually be made. If the recommendations presented in this thesis are pursued and implemented, it is highly probable that the resulting version of OSMagic will show that such conversion efforts are warranted. Proper response to the lessons learned in experimental conversions like ours will greatly reduce the time and effort required in actual conversions of computer-aided design tools.

Appendix A. *Using Three Interfaces to ObjectStore*

Concerns were raised in responses to the work by Jacobs (8) as to why OSMagic used ObjectStore's DML interface as opposed to its C Library interface. These responses questioned whether or not ObjectStore's C interface would have been better choice since Magic is written in C. Since the interface was to be changed anyway in order to use ObjectCenter, it was felt that it wouldn't require much more effort to code the OSMagic application so it could be compiled and linked using any one of the three ObjectStore interfaces. This would allow us to use the ObjectCenter environment and to explore issues concerning which interface is best suited to our application. This effort was not as straightforward as anticipated, however. In ObjectStore, database roots are defined in order to provide access points into the database which are then used to traverse to any desired data objects. Since we want OSMagic to access the same database when set up in any of the three interfaces, the same database roots have to be accessible via any of the interfaces. This proved to be a difficult task because ObjectStore's DML interface adds an extra level of indirection when it automatically defines database roots and persistent variables.

```
/* Straight DML way (OLD):
 *      persistent<magicdb1> osHashTable *dbCellDefTable = NULL;
 *
 * New way:
 */

#ifdef _OSMAGIC_DML

   persistent<magicdb1> osHashTable REALdbCellDefTable = {0};
   osHashTable *dbCellDefTable = NULL;

#else  /* Same for both the C and C++ Interfaces */

   osHashTable *dbCellDefTable = NULL;

#endif
```

Figure 10. Code modifications for definitions of database roots

```
#ifdef _OSMAGIC_DML
  database_root* root1 = magicdb1->find_root("REALdbCellDefTable");
  dbCellDefTable = (osHashTable*) root1->get_value();
#endif _OSMAGIC_DML

#ifdef _OSMAGIC_CLI  /* C Library Interface */
  database_root* root1 = database_root_find("REALdbCellDefTable",magicdb1);
  if (!root1) {
    dbCellDefTable = objectstore_alloc(osHashTable_type, 1, magicdb1);
    database_root* root1 = database_create_root(magicdb1,
                                                "REALdbCellDefTable");
    database_root_set_value(root1, dbCellDefTable, osHashTable_type);
  }
  else
    dbCellDefTable = (osHashTable*) database_root_get_value(root1);
#endif _OSMAGIC_CLI

#ifdef _OSMAGIC_CCLI /* C++ Library Interface */
  database_root* root1 = magicdb1->find_root("REALdbCellDefTable");

  if (!root1) {
    dbCellDefTable = new(magicdb1, 1, osHashTable_type) osHashTable;
    database_root* root1 = magicdb1->create_root("REALdbCellDefTable");
    root1->set_value(dbCellDefTable, osHashTable_type);
  }
  else
    dbCellDefTable = (osHashTable*) root1->get_value(osHashTable_type);
#endif _OSMAGIC_CCLI
```

Figure 11. Code for creating and reestablishing database roots

OSMagic uses the pointer to the hash table of persistent cell definitions as one of the roots into the database. Normally in a DML interface this pointer is defined as a persistent variable. The DML will then automatically create a database root with the same name when the pointer is first dereferenced. It is in how the DML does this that the extra level of indirection is introduced. To compensate for this, in order to use the same root when accessing the database via one of the other interfaces, the declaration of the pointer in the DML version has to be modified. Figure 10 shows how a root has to be defined in DML for it to correspond to the same root defined in either of the other interfaces. Note how the extra level of indirection in DML is handled. The database root is named *REALdbCellDefTable* and is accessible via the pointer named *dbCellDefTable*.

When using a DML persistent variable, its corresponding root is automatically resolved whenever the variable is referenced. When using the other interfaces the root pointer

```
#include <ostore/ostore.hh>
#include <ostore/coll.hh>
#include <ostore/relat.hh>
#include <ostore/manschem.hh>
#include "magic.h"
#include "geometry.h"
#include "tile.h"
#include "hash.h"
#include "database.h"

static void dummy () {
        OS_MARK_SCHEMA_TYPE(tile);
        OS_MARK_SCHEMA_TYPE(label);
        OS_MARK_SCHEMA_TYPE(celltilebody);
        OS_MARK_SCHEMA_TYPE(celluse);
        OS_MARK_SCHEMA_TYPE(Plane);
        OS_MARK_SCHEMA_TYPE(celldef);
        OS_MARK_SCHEMA_TYPE(osh1);
        OS_MARK_SCHEMA_TYPE(osh3);
        OS_MARK_SCHEMA_TYPE(char);
}
```

Figure 12. ObjectStore *schema.cc* file required for OSMagic

must be reestablished at the beginning of each transaction. In this case however, since it has been defined differently for compatibility with the C and C++ interfaces, the root must be reestablished in the DML interface now as well. Figure 11 depicts the code required to reestablish the root in each transaction depending on the interface used. This example also shows how the database root *REALdbCellDefTable* is created in the C and C++ Library interfaces.

When using DML, the ObjectStore version of the C++ compiler (OSCC) uses an update_schema option which ensures the compilation schema database gets updated while the application is being compiled. The C and C++ library interfaces require the creation of a **schema.cc** file. The schema file required by OSMagic is shown in Figure 12. ObjectStore uses this file to update the compilation schema database in batch mode. This file must have include statements for all header files which declare the persistent classes that will be used in the application. It must also have a dummy function with calls to the **OS_MARK_SCHEMA_TYPE()** macro for each of the persistent classes.

```
#ifdef _OSMAGIC_CCLI   /* for ObjectStore C++ Library Interface */
    os_typespec *osHashTable_type = new os_typespec ("osh3");
    os_typespec *CellDef_type = new os_typespec ("celldef");
    os_typespec *CellUse_type = new os_typespec ("celluse");
    os_typespec *Workspace_type = new os_typespec ("workspace");
    os_typespec *CellConfig_type = new os_typespec ("configuration");
    os_typespec *char_type = new os_typespec ("char");
#endif _OSMAGIC_CCLI

#ifdef _OSMAGIC_CLI   /* for ObjectStore C Library Interface */
    osHashTable_type = alloc_typespec ("osh3", 0);
    CellDef_type = alloc_typespec ("celldef", 0);
    CellUse_type = alloc_typespec ("celluse", 0);
    Workspace_type = alloc_typespec ("workspace", 0);
    CellConfig_type = alloc_typespec ("configuration", 0);
    char_type = alloc_typespec ("char", 0);
#endif _OSMAGIC_CLI
```

Figure 13. ObjectStore type specifications for persistent data types

Another area automatically handled by the DML interface is the defining of the type specifications required by ObjectStore for each data type stored in persistent memory. The C and C++ interfaces require the explicit declaration of these types specifications through the use of the ObjectStore **os_typespecs** construct. Figure 13 shows some examples of these typespecs as defined in each ObjectStore interface. These os_typespecs are then used in the allocation instructions for persistent memory. Remember the declaration of such type specifications is automatic in the DML interface.

As can be seen in the previous examples, preprocessor variables were used to code the three interfaces. Code was implemented in each interface for every ObjectStore instruction. Another example of the interface differences is shown in Figure 14. This example shows allocation of persistent memory space in each of the three interfaces. It shows this for regular allocation or for allocation in a configuration. An interesting note here is that the C and C++ library interfaces are very similar in syntax in most cases. However there are some instructions which are not available in the C Library interface. The `configuration::of`

```
#ifdef _OSMAGIC_CLI  /* C Library Interface */
    if ((cellConfig = configuration_of(cellDef)) != 0)
      cellUse = objectstore_alloc_in_config (CellUse_type, 1, cellConfig);
    else
      cellUse = objectstore_alloc (CellUse_type, 1, magicdb1);
#endif _OSMAGIC_CLI

#ifdef _OSMAGIC_CCLI  /* C++ Library Interface */
    if ((cellConfig = configuration::of(cellDef)) != 0)
      cellUse = new(magicdb1, cellConfig,  1, CellUse_type) CellUse;
    else
      cellUse = new(magicdb1, 1, CellUse_type) CellUse;
#endif _OSMAGIC_CCLI

#ifdef _OSMAGIC_DML
    if ((cellConfig = configuration::of(cellDef)) != 0)
      cellUse = new(magicdb1, cellConfig) CellUse;
    else
      cellUse = new(magicdb1) CellUse;
#endif _OSMAGIC_DML
```

Figure 14. Memory allocation in the three ObjectStore interfaces

instruction in this example is one such case. To get the capability of this instruction when using the C library interface, the C++ version of the instruction must be encapsulated in a binding as follows:

```
extern "C" configuration *configuration_of(void *ptr);

configuration *configuration_of(void *ptr) {
   return (configuration::of(ptr));
}
```

With similar code implemented for all ObjectStore instructions, OSMagic can then be set up to use any of the three interfaces by using the `-D` compiler option to define the preprocessor variable representing the interface desired at compile time. This allows for testing the application using each interface and the same database. Makefile changes and the creation of a **schema.cc** file are also required for the C and C++ interfaces.

Appendix B. *ObjectCenter Interface Issues*

1. Required Makefile Changes.

The following are the makefile changes required for the OSMagic application to be loaded into the ObjectCenter environment using a makefile.

```
# ObjectCenter Vars and rules
OCTR_INCS = -I$(OS_ROOTDIR)/include
OCTR_FLAGS = $(OCTR_INCS) -g -DDEBUG -D\_\_CPLUSPLUS

ocenter_osmagic_objs: ocenter_schema_standin
        #setopt load_flags $(OCTR_FLAGS)
        #load  ${LNKS} osmagicTop.o .os_schema.o bzero.o ${LIBS} ${LDLIBS}
        #link
        #source init_for_OC_cmode

ocenter_schema_standin: OSschema.cc
        $(OS_PRELNK) .os_schema.cc $(OS_COMPILATION_SCHEMA_DB_PATH) \
            $(OS_APPLICATION_SCHEMA_DB_PATH) ${LDLIBS}
        $(CC) -c .os_schema.cc
        touch ocenter_schema_standin
```

2. ObjectCenter Init File.

The following is the .ocenterinit file required for the OSMagic application.

```
/*  This init file set up so ObjectCenter and ObjectStore can work
    together in my environment.
*/

/*  Set the environment variable OS_RESERVE_AS to work around a problem
    in ObjectCenter where it is confused by system calls to munmap. The
    client avoids use of munmap and uses an alternate technique to manage
    address space instead. Use of this feature will cause some performance
    degradation in certain circumstances, so be careful.

    If you fail to set this environment variable you will get incorrect
    runtime errors from ObjectCenter complaining about valid persistent
    addresses, typically after at least one transaction has been completed
    and a subsequent transaction is initiated.
*/
```

```
setenv OS_RESERVE_AS yes


/* Suppress warning 11, this is the warning that complains about use
   of the old interface names with the "saber_" prefix rather than the
   new names with the "objectcenter_" prefix.
*/

suppress 11

ignore SIGSEGV

#define __cfront21 1

/* In order to have ObjectCenter Swap command use "path" option
   when looking for files I set the following option.  Note:  use
   the -C option with the swap command when loading C files.  Also
   can use -G option to load object code without debugging info.
   See page 4-16 of Using ObjectCenter book.
*/

setopt swap_uses_path

/* In order for ObjectCenter to reconize the .cc suffix as a C++
   file need to set the cxx.suffix option.  ObjectCenter will then
   look first for files with this suffix, if it doesn't find one
   it will then look for files with .C suffix and then with .c suffix.
*/

setopt cxx_suffix cc

'* In order to have proper memory to run OSmagic need to increase the
   ObjectCenter sbrk_size option.
*/

setopt sbrk_size 2048576

3. Cmode Actions File.  Envoked by makefile target.

/*  The interface that we (CenterLine) use to obtain vtbl addresses
    has been broken in v1.1.0 so that it only works when ObjectCenter
    is in cmode. You can deal with this problem by always making sure
    that you are in cmode before typing the "run" command. You can
```

```
    revert to "cxxmode" anytime after execution has been initiated.

    You can automate this process somewhat by defining the following
    "actions":

    a) action at _os_debugger_init__Fv
    { centerline_cmode("") ; }

    b) action at init_discriminants
    { centerline_cxxmode("") ; }

    Remember that they can only be defined after the functions are defined
    in the ObjectCenter workspace, ie. the program has been run at least
    once.

    Actually, this can be further automated by adding an extra line to the
    makefile - see example in this directory.  The file init_for_OC_cmode
    contains the above actions and is sourced by the makefile.

    If you fail to do this vtbls won't be swizzled correctly, and any
    attempt to access an existing database will cause the application to
    fail due to an incorrect vtbl address.
*/


action at _os_debugger_init__Fv
{ centerline_cmode(""); }
action at init_discriminants
{ centerline_cxxmode(""); }
```

Appendix C. *Persistent Declarations of Internal Work Objects*

While testing the implementation of our initial versioning model, a fatal error was encountered which stated that an attempt was made to access a frozen version of a data object. Analysis showed that the error occurred in the function that updates Magic windows with the results of the most recently issued command. In this case OSMagic was in the initialization transaction and the update window function was attempting to display the UNNAMED cell definition in the main window. The UNNAMED cell definition is always loaded when the window is to be in its initial state. Further analysis found that there are many temporary cell definitions, used internally by Magic in the implementation of its algorithms, which OSMagic had allocated persistently when they should have remained transient.

C.1 Problem Analysis

When the initial versioning model was implemented in OSMagic, it made configurations of each persistently defined cell definition. This is discussed in detail in Section 3.3.3 of the main text. When new configurations are created and loaded into the database, they are initially checked-in to the global workspace. The initial analysis found that the error was caused by the fact that the UNNAMED cell definition was checked into the global workspace when it was created but was not checked out during the initialization process. Therefore code was implemented to correct this problem. However the initialization process didn't progress much farther before finding a similar error on another cell definition.

A deeper analysis found that Magic defines and uses many temporary work cells in its algorithms for implementing its available commands. For example, Magic's routing algorithm could use 20 or more temporary cell definitions depending on the type of routing done and the options used. Magic designates these temporary cell definitions as *internal* by setting the CDINTERNAL flag bit in the cell definition's cd_flags mask. Most of these internal cell definitions are created and initialized by functions called during the start-up phase of Magic. Others are created when needed and deleted afterwards.

In Magic these internal cells only last for the life of the process and are never written or saved to disk. Any persistent data created during an algorithm is saved via persistent cell definitions by means of a copying function or by direct association between an internal cell definition and a persistent cell definition. Magic insures internal cell definitions are not saved to disk by checking for the CDINTERNAL flag bit. If the bit is set the cell is not saved.

The first OSMagic made all cell definitions persistent, including the *internal* ones. The functions where new cell definitions are created and defined were changed to allocate the new cells via ObjectStore's persistent new operator. No distinction was made between those which were temporary and those which were to be saved. The problem with this implementation became evident when the configurations needed for versioning were implemented for persistent cell definitions and thus the internal definitions as well. In the initial implementation of the basic versioning model, all persistent and configured cell definitions are initially checked-in to the global workspace of the database and are only checked out during *read* operations. Because of this, the internal cell definitions were checked-in but never checked-out. This caused the *attempted access of frozen version* errors. The problem didn't surface sooner since the versioning code had not been implemented. Also, since all of Magic's commands were not tested in the first effort, the problems the internal cells would have encountered weren't discovered.

The analysis of how Magic uses its internal cell definitions made it clear that they should not be persistently allocated in OSMagic, but rather transiently allocated as in Magic. This will insure they go away at the end of the process and are recreated each time the tool is used. There is no reason for saving these cells in the database since the information they hold is of no use. To do so without special provisions for reinitializing them would only introduce errors in the tool's algorithms.

C.2 Solution Analysis and Implementation

There are three classes of cell definitions used in a Magic session which OSMagic must address. First there are temporary work cells which are only used internally. These cells are never saved to files in Magic and therefore do not require persistence. They should

be allocated in transient memory as they were in the original Magic. To allocate them persistently complicates the code and makes it less efficient because of the I/O operations they require. Second are the cell definitions which must be persistently allocated and configured. These are the designs and definitions created and manipulated by OSMagic and saved in the database. They must be configured so their design versions can be tracked and used for the implementation of Magic commands and for control of the added multi-user capabilities the database brings to the tool.

The third class consists of cell definitions which are transient in nature but may contain data which could eventually be stored in persistent memory space. Because they are internal work cells, they should not remain defined when the process ends. However, since the data they contain can be, and often is, saved to persistent memory, special handling of these definitions or the data they contain is required. There are several ways to handle these kinds of cells. Probably the best approach is to treat them as any other internal transient cell definition and appropriately modify the commands that assign any data in them to persistent space. One example of how this would work is the UNNAMED cell definition which is the internal definition used when the edit window is placed in its initial state. Normally empty, it is in this cell where designers create new designs which, if desired, can be saved in the database. In Magic this is accomplished by copying the data from the UNNAMED cell definition to a newly allocated cell definition with a user-specified name. Since everything is transient in Magic, the new pointers are just set to what the old pointers are assigned. This new definition is then used to save the design to disk and the UNNAMED cell is cleared out for future use in the current session. In a similar way, one approach to dealing with this class of cells in OSMagic is to allocate them in transient memory, like the other internal definitions, and modify the copy routines to appropriately allocate persistent memory when objects created in *transient* cells are saved to persistent space. With this approach, the UNNAMED cell definition is allocated in transient memory and the cell definition for the new design is allocated and configured in persistent space. Then the copy routine should be modified to allocate new persistent space for each transient object pointed to by the *unnamed* cell definition and then assign those new objects to the pointers in the newly allocated *persistent* cell definition. There are

other internal cell definitions, like those used for the `select` command, in which only some of the data, not all of it, can be assigned to a persistent pointer. These cases each require handling according to the situations involved. Another approach to handling this class of cell definitions in OSMagic is to allocate them in persistent memory space so that any data objects created in them are persistent. These cells should not be configured for versioning purposes, however, since such cells are never stored in the database. Instead these cell definitions should be reinitialized or recreated at appropriate times. This approach then allows the commands to work essentially the same as in the original Magic. The first approach is clearly superior since it defines a more efficient and natural implementation of this class of cells. The persistent allocation of temporary data structures, as defined by the second approach, deviates from natural logic and is less efficient since it requires database access for each reference to those structures. However, because it required fewer changes, we chose the second approach for temporary implementation in the current prototype because of the time limitations we faced.

The original Magic tracks cell definitions by means of a hash table. Whenever a cell definition is created, whether for a design read in from disk or for the creation of an internal work cell, its name is stored in the table so the definition won't be created more than once in a session. The first OSMagic made this hash table persistent so that a cell definition is only created when initially loaded into the database. When configurations were added, a persistent hash table was created for tracking them as well. To implement the creation and tracking of both persistent and transient cell definitions requires another hash table for the transient cases. This requires modifications to the hash functions to allow them to create and manipulate the appropriate hash table, transient or persistent. A new parameter was added to the hash table initialization functions. This parameter is a boolean flag which indicates whether the table is to be created in persistent or transient memory. Figure 15 shows how this flag is used to insure a new table is properly allocated. New entries to the tables are allocated in a similar manner except that ObjectStore commands are used to determine how the table receiving the entry was allocated so that the new entry can be allocated in the same way.

```
  if (Persistent)

#ifdef _OSMAGIC_CLI  /* C Library Interface */
    if ((cellConfig = configuration_of(table)) != 0)
      table->ht_table = objectstore_alloc_in_config
                          (osHashEntry_type, table->ht_size, cellConfig);
    else
      table->ht_table = objectstore_alloc
                            (osHashEntry_type, table->ht_size, magicdb1);
#endif _OSMAGIC_CLI

#ifdef _OSMAGIC_CCLI   /* C++ Library Interface */
    if ((cellConfig = configuration::of(table)) != 0)
      table->ht_table =
         new(magicdb1, cellConfig, table->ht_size, osHashEntry_ptrtype)
                                                 osHashEntry*[table->ht_size];
    else
      table->ht_table = new(magicdb1, table->ht_size, osHashEntry_ptrtype)
                                                 osHashEntry*[table->ht_size];
#endif _OSMAGIC_CCLI

#ifdef _OSMAGIC_DML
    if ((cellConfig = configuration::of(table)) != 0)
      table->ht_table =
                  new(magicdb1, cellConfig) osHashEntry*[table->ht_size];
    else
      table->ht_table = new(magicdb1) osHashEntry*[table->ht_size];
#endif _OSMAGIC_DML

  else
    MALLOC(osHashEntry **, table->ht_table,
                  (unsigned) (sizeof (osHashEntry *)  * table->ht_size));
```

Figure 15. Allocation of cell definition hash tables

Magic allocates all cell definitions in transient memory. The first OSMagic allocated them all in persistent memory. To make OSMagic handle both, changes were required to all functions which create and initialize the new cell definitions and all their components. Like the hash table functions, these functions required changes which allowed for the distinction between the transient and persistent cases of cell definitions. They also required changes to tell whether a persistent cell definition should be configured or not. For the functions **DBCellNewDef** and **DBCellDefAlloc**, where objects are first created, a new boolean parameter was added to indicate whether to allocate the new definition in persistent or transient memory. The function **DBCellNewDef** also required a parameter indicating whether or not to configure a new persistent cell definition. These changes are on the same order as those

```
#ifdef _OSMAGIC_CLI  /* C Library Interface */
  if (objectstore_is_persistent(cellDef))
    if ((cellConfig = configuration_of(cellDef)) != 0)
      cellUse = objectstore_alloc_in_config (CellUse_type, 1,
                                                    cellConfig);
    else
      cellUse = objectstore_alloc (CellUse_type, 1, magicdb1);
  else
    MALLOC(CellUse *, cellUse, sizeof (CellUse));
#endif _OSMAGIC_CLI

#ifdef _OSMAGIC_CCLI   /* C++ Library Interface */
  if (objectstore::is_persistent(cellDef))
    if ((cellConfig = configuration::of(cellDef)) != 0)
      cellUse = new(magicdb1, cellConfig,  1, CellUse_type) CellUse;
    else
      cellUse = new(magicdb1, 1, CellUse_type) CellUse;
  else
    MALLOC(CellUse *, cellUse, sizeof (CellUse));
#endif _OSMAGIC_CCLI

#ifdef _OSMAGIC_DML
  if (objectstore::is_persistent(cellDef))
    if ((cellConfig = configuration::of(cellDef)) != 0)
      cellUse = new(magicdb1, cellConfig) CellUse;
    else
      cellUse = new(magicdb1) CellUse;
  else
    MALLOC(CellUse *, cellUse, sizeof (CellUse));
#endif _OSMAGIC_DML
```

Figure 16. Use of ObjectStore commands for determining state of objects

shown in Figure 15 for the hash functions. Changes were also required in those routines allocating space for the components of new or changed cell definitions. Whenever possible this was done by using the ObjectStore commands `objectstore::is_persistent`, to determine how to allocate the component, and `configuration::of`, to determine which, if any, configuration in which to allocate persistent components. Figure 16 shows an example of how these commands are used when creating new cell uses for a cell definition. The example shows how these commands are used in all three ObjectStore interfaces. In all, 56 functions in 26 files of 15 OSMagic modules required modifications due to these changes.

Appendix D. *Transient Addresses in Database - Detailed Discussion*

Probably the most coding intensive problem we encountered in our conversion effort was that of transient addresses in the database. For various reasons the initial OSMagic prototype did not properly allocate all of the component fields in OSMagic's persistent hash tables and cell definitions. Because these component fields were **not** allocated in persistent memory when created and initialized, transient addresses were stored in the database for them. ObjectStore defines such addresses as illegal, which when found will cause fatal errors. This problem occurred in the *ti_client,* and *ti_body* fields of the **Tile** data structure; the *cu_client* field of the **CellUse** data structure; the unassigned pointers of the **osHashTable** data structure; and the *infinityTile* which is assigned to certain tiles of the **Plane** data structure.

The initial identification of this problem was delayed because the errors being reported differed depending on which bad field was encountered. The field encountered differed based on how testing was being conducted and any attempted fixes to previous errors. For example, to insure the versioning code would properly check a version of a cell definition out of the global workspace into the user workspace, the load command was tested on the cell definition for **drfmchip**, a VLSI chip design used in our testing. During this test ObjectStore reported an attempted dereference of a transient address as a fatal error. Analysis at that point found the *cu_client* field was transiently allocated. After an attempted fix, the test was run again. This time ObjectStore reported an illegal pointer in the database which was caused by the *infinityTile.* Adding to the confusion was the fact that the smaller tutorial chips were loaded without complaint even though when viewing the data via the ObjectStore browser, transient addresses were found in the same places of all cell definitions. Because of this inconsistency it was not clear just what impact the transient addresses were having on the problems being reported. It was only after days of analysis and testing and communication with the support branch of Object Design Inc., that the impact of the addresses became clear. When resolving references to persistent data, ObjectStore has the option of optimizing out checks for illegal addresses. However, when dealing with larger data objects, a relocation algorithm, internal to ObjectStore, is invoked which does not optimize out these checks. Since the **drfmchip** cell is large,

this relocation algorithm is apparently invoked when loading it, and thus the checks for illegal addresses are done. When loading the much smaller tutorial cells, however, the relocation algorithm is not invoked and thus the checks are optimized out. Of course the question that immediately came to mind was why didn't this problem occur in the first OSMagic. Apparently since the cells in the first OSMagic were not being checked in and out of workspaces, thereby not requiring the floating of necessary pointers by ObjectStore's memory mapping algorithm, the internal relocation algorithm was not invoked and the checks for illegal pointers were optimized out.

Once the problem was clarified, the next step was to identify all areas in the database where transient addresses appeared and implement the necessary fixes. Illegal pointers can be found by using the ObjectStore `database::set_check_illegal_pointers(1)` option. Setting this option to **1** forces ObjectStore to check for illegal pointers in all segments of the database. The default setting of this option is **0** which gives ObjectStore the option of making the checks or optimizing them out. By explicitly setting this option to **1** and using the ObjectStore database browser, all components in the database with transient addresses were identified. Some were easy to fix, while others were not. Just how each area was addressed is detailed in the following sections.

D.1 The Cu_client Field

The **CellUse** data structure has a field called *cu_client* which, according to comments in the code, is *"space for rent"*. This field is declared to be of type `ClientData`, where `ClientData` is a `typedef` for `char*` (pointer to character string). Analysis found that the only values ever assigned to the *cu_client* field were integers which were cast as `ClientData` when assigned to the filed. In some cases when the value in this field is tested, it is first cast back as an integer, while in other cases the integer being tested against is first cast as `ClientData`. In all cases, only integer data is being manipulated. For that reason the fix was easy, requiring changes to just one header file and several functions. The *cu_client* field of the **CellUse** data structure was changed to be of type integer and all casts were removed from the references to the field and from those values tested against the field.

D.2 Unused Hash Table Pointers

When a persistent hash table was created and initialized its pointers were set to a defined value called NIL. They retained this value until assigned the address of an entry added to the table. When looking for transient addresses via the database browser, the unused pointers did not explicitly say *transient address* as in the other cases, but instead showed a value which at first glance appeared to be a proper value. However, this value was interpreted by ObjectStore as a transient address. Closer analysis found that NIL was defined as an **osHashEntry** pointer casting of the integer 1 shifted left 29 times. This produces a value which, when stored in the database, is seen as a transient address. Comments in the code indicated that the reason for defining NIL in this manner was to guarantee the pointer would be seen as invalid and thus cause a core dump if an errant attempt was made to indirect through it. The predefined variable NIL is used in the hash table functions to test the status of the pointers in the various algorithms which search through the tables. Analysis showed that, if implemented right, the use of the NULL pointer would serve the same purpose and solve the transient address problem. Therefore the fix requiring the least amount of changes was to redefine the NIL preprocessor variable as NULL, the C value for a NULL pointer.

D.3 The InfinityTile

The infinityTile is a default tile used when allocating and initializing a new **Plane** data structure. It is used to insure boundary tiles, which are important in the corner stitching Magic uses, never have zero width or height. When the problem with transient and persistent cell definitions was solved, the allocation of this tile was set up to be persistent for persistent Planes and transient for transient Planes. However, it was not noticed at that point that the pointer to the infinityTile was declared as a static variable so that the tile, which was constant for the duration of the process, would only be allocated once. Since during the initialization phase of OSMagic a temporary transient cell definition is allocated first, the infinityTile was allocated transiently. Later when a persistent cell was allocated, the transient infinityTile was assigned in numerous places in numerous Planes via a recursive algorithm.

To solve this problem, two static pointers were defined. One for a persistent *PinfinityTile* and one for a transient *TinfinityTile*. Then the appropriate infinityTile was assigned based on how the cell definition, for which the Plane was being created, was allocated.

D.4 The Ti_body and Ti_client Fields

The *ti_body* and *ti_client* fields are found in the **Tile** data structure. Both were declared to be of the character pointer type, `ClientData`. Comments indicate the *ti_body* field is the *body* of a tile. Analysis found, however, that this has different meanings for different types of tiles. When a tile is part of a *paint plane*, its *ti_body* field is assigned an integer value which indicates the type of paint, if any, assigned to that tile. When a tile is part of a *cell plane* the *ti_body* field is assigned a pointer to a CellTileBody structure which defines a list of cell uses that appear in the area covered by that tile. There are other uses of this field as well. To indicate a *space* tile the value TT_SPACE, predefined as integer 0, is assigned to the field, and in the routing algorithms the field is assigned predefined integers, like CHAN_HRIVER or CHAN_BLOCKED, which indicate the type of routing channel the tile represents. Depending on the case, the field is initialized to NULL, zero, or -1, so searching algorithms can determine a tile's status. The *ti_body* field can be used, therefore, to hold different values, which can be of different data types, depending on how the tile is to be used. The original Magic, and by default the first OSMagic, implemented this by declaring the *ti_body* field as type `ClientData`, and then casting all values assigned to the field with the same type. Then whenever referenced, the field was cast back to the appropriate data type, depending on the use it was expected to have.

The *ti_client* field can also be assigned values of different data types and the same method of allowing this is used in the original Magic. However, the *ti_client* field doesn't have as defined a purpose as *ti_body*. It is used as a general purpose utility field by many of the application's algorithms and as such can hold values of many different data types. But at the end of each algorithm the field is returned to its default value of MINFINITY, a predefined integer value. This default value is used often to determine the status of the *ti_client* field, especially in the many search algorithms. An example of the use of this field

can be seen in the selection algorithm for area and regions, where the area or region data structures are assigned to *ti_client* by first casting them as `ClientData`.

Because these fields were defined as pointer types, they held addresses as their values. This was true even when they were assigned integers since everything assigned to the fields was first cast as `ClientData`, the defined pointer to character data type. The purpose behind the *casting* approach used in the original Magic was to use the same memory space to hold the different data types these fields could represent. While not good programming practice, this approach will work in transient memory space as long as the appropriate casting is used to return the field to its original data type before use. This approach will not work, however, when these fields are part of data structures which are allocated in persistent memory space. Pointer fields in such structures must be assigned addresses to persistent memory space that is allocated for a specific data type. The use of casting to assign data types different from that allocated for a persistent field just won't work. Instead, a means must be implemented which allows these fields to hold different data types such that ObjectStore knows what type of data the fields are suppose to have for any particular reference.

The C **union** is the proper construct to use when different types of data are to be stored in the same memory space. As with C **structures** a union can be defined to have numerous fields of different types. The difference here is that only one field of a union is valid at any one time. It is up to the application to keep straight what data the union holds for any particular reference. When space is allocated for a union, it is allocated for the size of the largest data type the union can represent.

To correct the problem in OSMagic with transient data in the *ti_body* and *ti_client* fields, the **Tile** data structure was changed by defining each of these fields as C unions and adding tags for each union. The tags are used to indicate which field of a union is valid for any particular reference. Then discriminant functions were defined for each union which return the value of the tag representing that union. These functions are required by ObjectStore so it can determine the actual layout of the persistent object represented by the union when mapping it into memory. Finally, all references to either *ti_body* or

ti_client were changed to access the appropriate fields of the unions, and, in the case of assignments, to insure the tags were properly assigned.

To implement this solution, we first had to determine what data types each union could have. Analysis found that the *ti_body* union could have either integer or CellTileBody pointers as data types. The *ti_client* field on the other hand could have many different data types. A proper implementation of the union construct for this field would require major changes. However, since the field is always returned to the default integer value MINFINITY, that's the only data type ever stored in the database for this union. This allowed us to implement the less than optimal solution of defining the union as having just two data types, integer and `ClientData`. The discriminant function for this union is then defined such that it always returns a tag representing the integer field of the union. In this way ObjectStore will always view the union layout as being of the integer data type and never try to interpret the pointer field of the union. This allows us to use the same method of casting the different structures to `ClientData` when they are assigned to the pointer field of the *ti_client* union. The new definition for a **Tile** data structure is shown in Figure 17.

The new **Tile** structure also includes declarations of the discriminant functions for C++ modules. Because the OSMagic application is a mixture of C++ and C modules, the discriminant functions must also be defined in their corresponding C notation. This is accomplished by determining the mangled name of the corresponding C++ function and declaring the discriminant functions with that name in a top level C module as follows:

```
int discriminant_body_union__4tileFv(this_tile)
   Tile *this_tile;
   {
       return this_tile->body_union_tag;
   }
int discriminant_client_union__4tileFv() {return 1;}
```

They are then defined as external functions in the appropriate header file.

By far, the hardest part of this fix was the changing of all references and assignments to the *ti_body* and *ti_client* fields. There are hundreds of such references which, in many cases, are complicated by the use of C macros and numerous levels of indirect references allowed by the liberal use of casting. Compounding the problem is the nearly complete

```
typedef struct tile
{
  int body_union_tag;   /* Tag for type of data stored in ti_body */
  union body_union
  {
     struct celltilebody *tileBody;  /* when using body structure */
     int tileBodyType;              /* used for integer tile type */
  } ti_body;
#ifdef __cplusplus        /* Required by ObjectStore */
  int discriminant_body_union() {return body_union_tag;}
#endif
  struct tile *ti_lb;          /* Left bottom corner stitch */
  struct tile *ti_bl;          /* Bottom left corner stitch */
  struct tile *ti_tr;          /* Top right corner stitch */
  struct tile *ti_rt;          /* Right top corner stitch */
  Point        ti_ll;          /* Lower left coordinate */
  int client_union_tag;        /* Tag for type of data in ti_client */
  union client_union
  {
     int default_value;        /* Set to MINFINITY.  This field always
                                * returned to this value, so for the
                                * persistent case, this is only way
                                * the data is stored
                                */
     ClientData utility_ptr;   /* used when a function wants to used
                                * this field for a utility ptr.
                                */
  } ti_client;    /* This space for hire.  Warning: the default value
                   * for this field, to which all users should return
                   * it when done, is MINFINITY instead of NULL.
                   */
#ifdef __cplusplus       /* required by ObjectStore */
  int discriminant_client_union() {return 0;}
#endif
} Tile;
```

Figure 17. New Tile Structure Data Type

lack of structured programming in the original Magic. In many cases defined preprocessor variables are used as means of undocumented short cuts for referencing fields of data structures instead of the normal and clear, dot or pointer notation. Many of the macros used are implemented by calling other macros, which themselves call other macros, and so forth. It required days of analysis to determine where all the references to these fields occurred and how the references were made. In all 124 functions and headers required changes in 56 files of 14 OSMagic modules.

Appendix E. *Raw Performance Test Results*

Test on *drfmchip* using Original Magic

Command	CPU Time	Elapsed Time	Page Faults		Disk Blocks	
			with I/O	w/o I/O	In	Out
initialize	5.88	13.97	101	53	20	1
	6.05	14.43	101	53	20	1
	5.78	9.58	3	150	0	1
	5.69	9.60	20	174	20	1
	5.79	9.75	0	184	1	1
	5.74	9.74	20	174	20	1
load drfmchip	0.09	0.29	5	64	0	0
	0.08	0.23	3	0	0	0
	0.08	0.10	0	8	0	0
	0.08	0.08	0	8	0	0
	0.06	0.08	0	8	0	0
	0.08	0.09	0	8	0	0
expand	6.87	17.75	146	104	146	0
	7.08	18.84	146	102	146	0
	7.15	17.15	146	104	146	0
	7.10	16.64	146	104	146	0
	7.24	17.39	146	102	146	0
	7.06	17.11	146	104	146	0
getcell test	0.06	2.43	0	0	0	1
nested 4	0.07	1.41	0	0	0	1
levels deep	0.08	1.38	0	47	0	1
in subcell	0.07	1.98	0	0	0	1
big_nandmux	0.08	1.40	0	0	0	1
	0.08	1.23	0	47	0	1
getcell test	0.06	1.65	4	11	1	1
nested 8	0.08	1.85	3	1	1	1
levels deep	0.07	1.01	0	5	0	1
in subcell	0.06	0.95	1	5	1	1
mcell10	0.08	1.91	1	5	1	1
	0.06	1.32	0	5	0	1

Test on *drfmchip* using OSMagic (32 MB RAM)
with all data in the global workspace

Command	CPU Time	Elapsed Time	Page Faults		Disk Blocks	
			with I/O	w/o I/O	In	Out
initialize	6.49	15.25	67	399	15	1
	6.65	17.79	111	391	18	1
	6.48	14.26	72	392	17	1
	6.47	13.41	37	444	9	1
	6.50	13.34	33	447	11	1
	6.51	13.30	51	443	6	1
load drfmchip	1.22	8.68	21	227	0	0
	1.16	3.32	4	252	0	0
	1.16	3.23	18	226	0	0
	1.10	3.36	11	245	0	0
	1.13	2.75	16	232	0	0
	1.20	4.18	11	245	0	0
expand	134.66	264.93	1031	3062	1	0
	135.90	280.18	1165	3497	1	0
	135.15	250.76	1097	3305	1	0
	134.64	265.97	761	3692	0	0
	135.53	271.19	745	4097	0	0
	133.98	290.77	587	4331	0	0
getcell test	1.03	1.99	10	128	0	0
nested 4	1.26	1.88	1	221	0	0
levels deep	1.25	2.12	6	214	0	0
in subcell	1.21	2.48	5	242	0	0
big_nandmux	1.23	2.88	7	206	0	0
	1.24	2.25	15	206	0	0
getcell test	1.62	3.80	22	260	0	0
nested 8	1.63	3.93	24	244	0	0
levels deep	1.63	3.42	13	248	0	0
in subcell	1.61	3.43	18	225	0	0
mcell10	1.61	3.59	21	247	0	0
	1.60	3.40	8	246	0	0

Test on *drfmchip* using OSMagic (32 MB RAM)
with all data in the user workspace

Command	CPU Time	Elapsed Time	Page Faults		Disk Blocks	
			with I/O	w/o I/O	In	Out
initialize	6.67	16.57	51	427	12	1
	6.71	14.14	7	456	1	1
	6.74	14.86	15	454	2	1
	6.71	13.45	8	454	1	1
	6.69	12.76	6	453	1	1
	6.86	13.19	3	461	1	1
load drfmchip	0.84	2.72	9	209	0	0
	0.89	3.22	0	221	0	0
	0.85	2.51	0	222	0	0
	0.87	1.79	0	222	0	0
	0.84	1.74	0	222	0	0
	0.88	1.84	0	223	0	0
expand	22.96	102.28	155	2925	1	0
	22.62	103.15	66	3089	1	0
	22.66	99.55	58	3075	0	0
	22.87	97.61	91	3028	0	0
	22.90	98.15	41	3071	0	0
	22.73	90.20	28	3086	0	0
getcell test	1.31	3.87	7	249	0	0
nested 4	1.26	2.18	10	218	0	0
levels deep	1.19	1.78	3	219	0	0
in subcell	1.24	1.63	2	220	0	0
big_nandmux	1.27	1.66	2	251	0	0
	1.21	1.72	1	251	0	0
getcell test	1.22	1.57	0	205	0	0
nested 8	1.15	1.68	0	208	0	0
levels deep	1.19	1.47	0	211	0	0
in subcell	1.21	1.55	0	211	0	0
mcell10	1.19	1.53	0	212	0	0
	1.16	1.51	0	212	0	0

Test on *tut4a* using Original Magic

Command	CPU Time	Elapsed Time	Page Faults		Disk Blocks	
			with I/O	w/o I/O	In	Out
initialize	5.77	9.87	19	173	19	1
	5.77	9.67	19	173	19	1
	5.67	9.42	22	173	19	1
	6.04	13.91	101	55	20	1
	5.73	9.51	20	173	20	1
	5.87	9.67	21	173	19	1
load tut4a	0.02	0.03	0	7	0	0
	0.02	0.03	0	7	0	0
	0.00	0.14	2	0	0	0
	0.02	0.05	1	7	1	0
	0.01	0.03	0	7	0	0
	0.03	0.02	0	7	0	0
expand	0.03	0.04	0	2	0	0
	0.03	0	2	2	2	0
	0.02	0.29	3	0	2	0
	0.04	0.15	2	2	2	0
	0.03	0.06	0	2	0	0
	0.04	0.06	0	2	0	0
getcell test	0.10	1.53	0	5	0	1
top level	0.08	2.00	1	5	1	1
	0.03	3.09	3	1	1	1
	0.03	1.59	1	5	1	1
	0.05	1.73	1	5	1	1
	0.04	1.51	1	5	1	1

Test on *tut4a* using OSmagic (32 MB RAM)
with all data in the global workspace

Command	CPU Time	Elapsed Time	Page Faults		Disk Blocks	
			with I/O	w/o I/O	In	Out
initialize	6.70	19.69	154	242	20	1
	6.69	22.15	171	241	23	1
	6.64	20.98	163	213	20	1
	6.71	20.77	147	248	20	1
	6.85	25.24	208	208	24	1
	6.76	22.78	169	214	21	1
load tut4a	0.58	1.87	9	78	0	0
	0.63	1.99	11	79	0	0
	0.63	1.66	7	85	0	0
	0.62	1.90	12	76	1	0
	0.61	2.00	6	86	0	0
	0.63	1.89	9	82	0	0
expand	0.98	2.33	5	125	0	0
	0.95	2.24	6	119	0	0
	0.98	2.65	2	128	0	0
	0.99	3.15	7	119	0	0
	0.95	2.67	3	127	0	0
	0.96	2.16	6	122	0	0
getcell test top level	0.96	2.95	4	127	0	0
	0.91	1.91	7	120	0	0
	0.92	1.93	2	129	0	0
	0.93	2.15	6	120	0	0
	0.96	1.80	2	130	0	0
	0.94	1.84	6	123	0	0

Test on *tut4a* using OSmagic (32 MB RAM)
with all data in the user workspace

Command	CPU Time	Elapsed Time	Page Faults		Disk Blocks	
			with I/O	w/o I/O	In	Out
load tut4a	0.65	1.83	1	83	0	0
	0.63	1.22	1	80	0	0
	0.69	1.48	3	82	0	0
	0.64	1.02	3	77	0	0
	0.67	1.38	0	89	0	0
	0.65	1.43	0	89	0	0
expand	0.62	1.07	0	80	0	0
	0.70	1.10	0	91	0	0
	0.76	1.12	0	90	0	0
	0.69	1.05	0	91	0	0
	0.72	0.97	0	92	0	0
	0.71	0.99	0	92	0	0
getcell test	0.89	1.28	0	106	0	0
top level	0.90	1.31	0	107	0	0
	0.91	1.21	0	107	0	0
	0.90	1.20	0	108	0	0
	0.91	1.29	0	108	0	0
	0.92	1.19	0	108	0	0

Test with *drfmchip* using OSMagic (48 MB RAM)
with all data in the global workspace

Command	CPU Time	Elapsed Time	Page Faults		Disk Blocks	
			with I/O	w/o I/O	In	Out
initialize	6.71	18.815	140	324	21	1
	6.74	19.385	140	321	20	1
	6.66	21.362	140	310	19	1
	6.64	19.744	140	324	19	1
	6.71	19.461	145	318	24	1
	6.77	20.384	147	302	20	1
load tut4a	0.64	2.149	5	92	0	0
	0.62	1.954	5	92	0	0
	0.62	2.320	5	92	0	0
	0.64	2.194	5	92	0	0
	0.67	2.378	5	92	0	0
	0.64	1.907	6	91	0	0
load drfmchip	1.18	2.889	0	258	0	0
	1.10	4.085	0	258	0	0
	1.13	3.435	1	257	0	0
	1.15	3.811	0	258	0	0
	1.18	3.787	0	258	0	0
	1.12	3.877	0	258	0	0
expand	128.44	299.222	22	4778	0	0
drfmchip	129.78	342.824	2	4792	0	0
	129.97	322.592	4	4790	0	0
	130.50	320.111	27	4800	0	0
	130.44	299.878	8	4786	0	0
	130.38	315.179	18	4776	0	0
getcell test	1.55	5.745	1	234	0	0
nested 8	1.58	4.876	1	272	0	0
levels deep	1.60	4.538	1	272	0	0
in subcell	1.59	4.546	1	272	0	0
mcell10	1.63	4.235	3	233	0	0
	1.64	3.877	1	272	0	0

Bibliography

1. Ahmed, Rafi and Shamkant B. Navathe. "Version Management of Composite Objects in CAD Databases." *ACM Proceedings of SIGMOD '91 International Conference on Management of Data, Denver Co.*. 218–227. June 1991.
2. Andrews, Tim, et al. "ONTOS: A Persistent Database for C++." *Object-Oriented Databases with Applications to CASE, Networks, and VLSI CAD* edited by Rajiv Gupta and Ellis Horowitz, chapter 21, 387–406, Englewood Cliffs, NJ: Prentice Hall, Inc., 1991.
3. Bancilhon, F., et al. "A Model of CAD Transactions." *Proceedings of Eleventh International Conference on Very Large Databases (VLDB)*. 25–33. August 1985.
4. Banerjee, J., et al. "Data Model Issues for Object-Oriented Applications." *Readings in Object-Oriented Database Systems* edited by Stanley B. Zdonik and David Maier, chapter 3, 197–208, San Mateo, CA: Morgan Kaufmann Publishers, Inc., 1990.
5. Beech, David and Brom Mahbod. "Generalized Version Control in an Object-Oriented Database." *Proceedings of Fourth International Conference on Data Engineering*. 14–22. February 1988.
6. Cattell, R.G.G. *Object Data Management, Object-Oriented and Extended Relational Database Systems*. Menlo Park, California: The Addison-Wesley Publishing Company, Inc., 1991.
7. Chou, H.T. and W. Kim. "A Unifying Framework for Version Control in a CAD Environment." *Proceedings of Twelfth International Conference on Very Large Databases (VLDB)*. 336–344. August 1986.
8. Jacobs, Capt Timothy M. *An Object-Oriented Database Implementation of the Magic VLSI Layout Design System*. MS thesis, AFIT/GCS/ENG/91D-09, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991.
9. Katz, Randy H. and Tobin J. Lehman. "Database Support for Versions and Alternatives of Large Design Files," *IEEE Transactions on Software Engineering*, *2*:191–200 (March 1984).
10. Katz, R.H. and E. Chang. "Managing Change in a Computer-Aided Design Database." *Proceedings of 13th International Conference on Very Large Databases (VLDB)*. 455–462. September 1987.
11. Kim, W., et al. "A Transaction Mechanism for Engineering Design Databases." *Proceedings of Tenth International Conference on Very Large Databases (VLDB)*. 355–362. August 1984.
12. Kitagawa, Hiroyuki and Nobuo Ohbo. "Design Data Modeling with Versioned Conceptual Configuration." *IEEE Proceedings of the 13th International Computer Software and Applications Conference*. 225–233. September 1989.

13. Korth, H.F. and A. Silberschatz. *Database System Concepts.* New York: McGraw-Hill Book Company, 1986.

14. Landis, Gordon S. "Design Evolution and History in an Object-Oriented CAD/CAM Database," *IEEE COMPCON*, 297-303 (March 1986).

15. Object Design, Inc., Burlington, Massachusetts. *ObjectStore User Guide*, March 1991.

Vita

Captain Gary M. Lightner was born on 19 January, 1954 in Baltimore, Maryland. He graduated from James Wood High School in Winchester, Virginia in 1972. He then entered the United States Air Force as an enlisted member in the fall of 1972. He entered Arizona State University via the Airman Education and Commissioning Program (AECP) in 1982 from which he graduated in 1985 with a Bachelor of Science Degree in Computer Science. He then attended Officer Training School at Lackland AFB, Texas where he received a commission as a Second Lieutenant in the United States Air Force. His first duty assignment as an officer was as a Cruise Missile Computer Systems Analyst for the Defense Communications Agency in Washington D.C., where he specialized in the design and management of the Terrain Contour Mapping (TERCOM) database functions. In late 1988 he relocated to the 7th Communications Group at the Pentagon where he managed information system software development for Headquarters United States Air Force, Deputy Chief of Staff, Productivity and Programs, Directorate of Programs and Evaluation. While there he served as an Air Staff Support Programmer Analyst for the Program Data System (PDS) in direct support of the Biennial Planning, Programming, and Budgeting System (BPPBS) and as a Headquarters Systems Replacement Program (HSRP) Project Officer responsible for the transition of the existing PDS to new software and hardware. He entered the School of Engineering, Air Force Institute of Technology in May, 1991.

Permanent address: 6589 Deer Bluff Drive
Huber Heights, Ohio 45424

REPORT DOCUMENTATION PAGE

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1992	Master's Thesis

4. TITLE AND SUBTITLE
VERSION AND TRANSACTION MANAGEMENT IN OSMAGIC: AN OBJECT-ORIENTED DATABASE IMPLEMENTATION OF THE MAGIC VLSI LAYOUT DESIGN TOOL

6. AUTHOR(S)
Gary M. Lightner, Capt, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

AFIT/GCS/ENG/92D-07

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
DARPA/CSTO
3701 North Fairfield Dr
Arlington, Va 22203-1714

ASC/RWWW
WPAFB OH 45433

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)
The goal of this thesis was to study the feasibility of using an object-oriented database management system to provide the functionality and performance needed to support a complex computer-aided design tool. We do this by modifying OSMagic, a prototype system of the Magic very large scale integrated (VLSI) circuit design tool implemented on the ObjectStore object-oriented database management system. OSMagic was changed to support three different interfaces to ObjectStore and to work in a networking environment. We then designed and examined the use of version and transaction management models as a means of addressing the weaknesses of the prototype as well as a means of introducing new multi-user capabilities. Throughout the effort errors discovered in data definition and allocation were corrected and lessons which pertain to all such conversions were noted. While the current OSMagic is not yet a fully functional version of Magic, the foundation is now in place for achieving this by implementing our more complex transaction model. OSMagic performs as well or better than Magic in about half of the areas tested. Those areas that showed performance degradation were examined with respect to the tradeoffs between added multi-user capabilities and database support versus the severity of the degradation, user perception of the degradation, and the frequency of use for each area. When these tradeoffs were considered along with the expected increased performance of a fully implemented and optimized OSMagic, we conclude that it is highly probable that object-oriented database management systems can provide the functionality and performance needed to support typical computer-aided design applications.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Object-Oriented, Database Management System, Computer-Aided Design, Very Large Scale Integration	107
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL